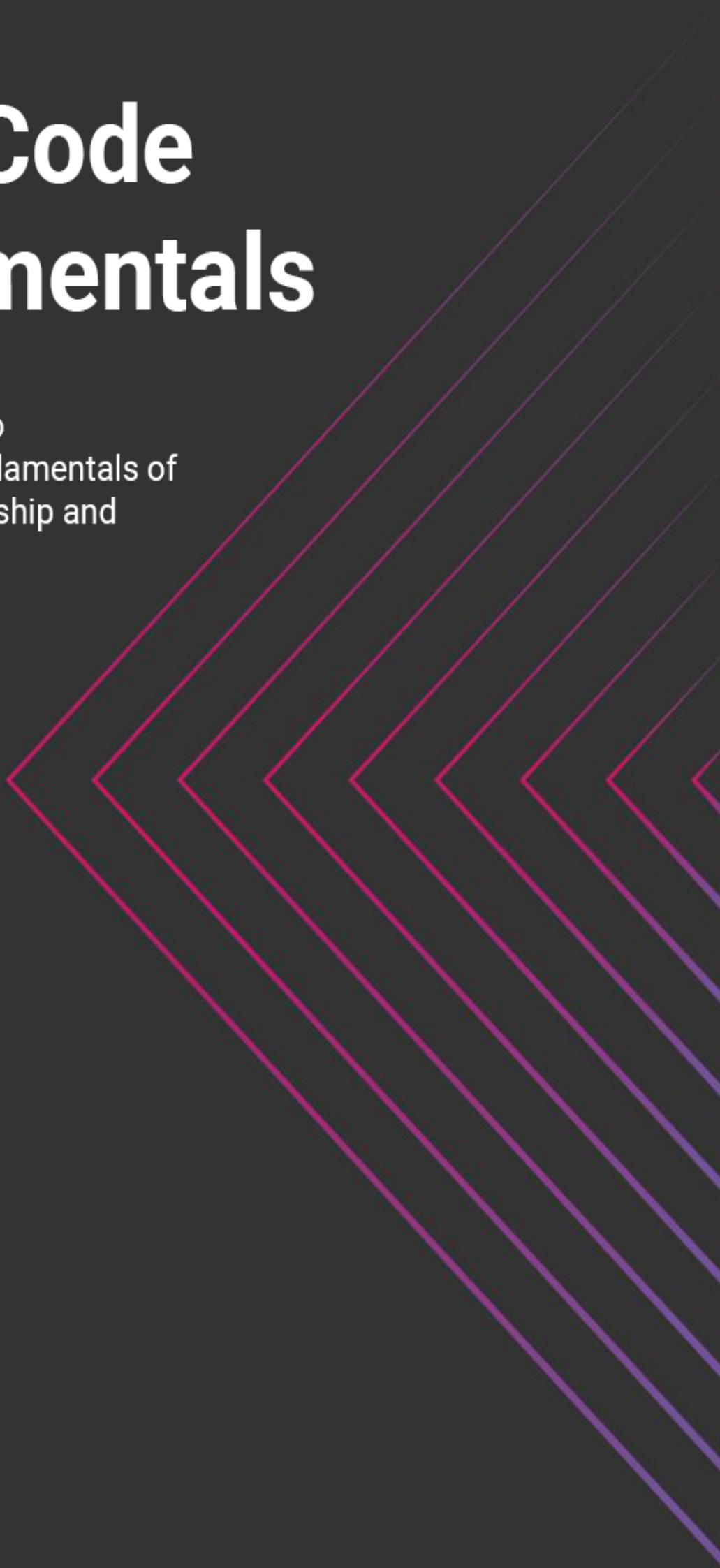
Clean Code Fundamentals

A **Hands-on Guide** to Understand the Fundamentals of Software Craftsmanship and Clean Code in Java



Martin Hock

Clean Code Fundamentals

Hands-on Guide to Understand the Fundamentals of Software Craftsmanship and Clean Code in Java

Martin Hock

This book is for sale at http://leanpub.com/clean-code-fundamentals

This version was published on 2024-02-08

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

© 2019 - 2024 Martin Hock

Tweet This Book!

Please help Martin Hock by spreading the word about this book on Twitter!

The suggested hashtag for this book is #CleanCodeFundamentals.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

#CleanCodeFundamentals

Contents

Preface . . . . . **i**
What is the goal of this book? i
Which topics can you expect? i
Who should read this book? ii
What you must learn about Software Development yourself? ii
What about the code examples and typographic conventions? iii
Which literature is this book based on? iii
Giving feedback? iv

1. Introduction to Software Craftsmanship and Clean Code **1**
1.1 A Passion for Software Development 1
1.2 Manifesto for Software Craftsmanship 4
1.3 Clean Code Developer 5
1.4 Boy Scout Rule 7
1.5 Broken Windows Theory 8
1.6 Cargo Cult Programming 9
1.7 Knowledge - Expertise 10

2. Basics of Software Design **11**
2.1 Software Design Pyramid 11
2.2 Basic concepts of OOD 12
2.3 Goals of Software Design 15
2.4 Symptoms of *bad* design 17
2.5 Criteria for *good* design 18
2.6 Information Hiding 19
2.7 Cohesion 21
2.8 Coupling 22
2.9 Cohesion - Coupling 23
2.10 Big Ball of Mud 24
2.11 Architecture Principles 25
2.12 Cognitive Psychology and Architectural Principles 26
2.13 Layered Architecture 27
2.13.1 Use of Layered Architecture 27
2.13.2 Violated Layered Architecture 29
2.13.3 Horizontal Layering 30
2.13.4 Feature-Based Layering - Single package 31
2.13.5 Feature-Based Layering - Slices before layers 33

2.13.6 Feature-Based Layering – Hexagonal Architecture 34
2.13.7 The Java Module System 35
2.14 Architecture Documentation 36
2.15 Testing the Architecture and Design 38
2.16 Software Engineering Values 40
2.17 Team Charter 41

3. Clean Code Best Practices 42
3.1 Communicate through code 42
3.1.1 Use Java code conventions and avoid misinformation 42
3.1.2 Choose an expressive name and avoid mental mapping 44
3.1.3 Make differences clear with meaningful variable names 44
3.1.4 Use pronounceable names 45
3.1.5 Do not hurt the readers 46
3.1.6 Don‘t add redundant context 47
3.1.7 Don’t add words without additional meaning 49
3.1.8 Don’t use *and* or *or* in method names 50
3.1.9 Use positive names for boolean variables and functions 51
3.1.10 Respect the order within classes 52
3.1.11 Group by line break 54
3.1.12 Prefer self-explanatory code instead of comments 55
3.1.13 Refactor step by step 57
3.2 Bad comments 58
3.2.1 Redundant comments 58
3.2.2 Misleading comments 58
3.2.3 Mandatory comments 59
3.2.4 Diary comments 59
3.2.5 Gossip 60
3.2.6 Position identifier 60
3.2.7 Write-ups and incidental remarks 61
3.2.8 Don’t leave commented out code in your codebase 61
3.2.9 Rules for commenting 61
3.3 Classes and objects 62
3.3.1 Classes 62
3.3.2 Functions 63
3.3.3 Variables 64
3.4 Shapes of code 65
3.4.1 Spikes 65
3.4.2 Paragraphs 65
3.4.3 Paragraphs with headers 66
3.4.4 Suspicious comments 66
3.4.5 Intensive use of an object 66

4. Software Quality Assurance 67
4.1 Test Pyramid 67
4.2 Test Classification 68
4.3 Test-driven Development (TDD) 69

4.4 Unit testing with JUnit 5 . . . 71
4.4.1 Unit Tests . . . 71
4.4.2 JUnit 5 . . . 71
4.4.3 First unit test . . . 73
4.4.4 Assertions . . . 74
4.4.4.1 `assertEquals()` / `assertArrayEquals()` . . . 74
4.4.4.2 `assertSame()` / `assertNotSame()` . . . 74
4.4.4.3 `assertTrue()` / `assertFalse()` / `assertAll()` . . . 75
4.4.4.4 `assertNull()` / `assertNotNull()` . . . 75
4.4.4.5 `assertThrows()` . . . 75
4.4.4.6 `assertTimeout()` . . . 76
4.4.4.7 `fail()` . . . 76
4.4.5 Annotations . . . 77
4.4.5.1 `@Test` . . . 78
4.4.5.2 `@BeforeEach / @AfterEach` . . . 79
4.4.5.3 `@BeforeAll / @AfterAll` . . . 80
4.4.5.4 `@Disabled` . . . 81
4.4.5.5 `@DisplayName` . . . 82
4.4.5.6 `@Tag` . . . 82
4.4.5.7 `@Timeout` . . . 83
4.4.6 Assumptions . . . 84
4.4.6.1 `assumeFalse()` . . . 84
4.4.6.2 `assumeTrue()` . . . 84
4.4.6.3 `assumingThat()` . . . 84
4.4.7 Parameterized Tests . . . 85
4.4.7.1 `@ValueSource` . . . 85
4.4.7.2 `@MethodSource` . . . 86
4.4.7.3 `@CsvSource` . . . 87
4.4.7.4 `@CsvFileSource` . . . 87
4.4.7.5 `@ArgumentsSource` . . . 88
4.5 More on Unit Tests . . . 89
4.5.1 Heuristics . . . 89
4.5.2 Naming of test methods . . . 92
4.5.3 Object Mother . . . 93
4.5.4 Test Data Builder . . . 94
4.5.5 F.I.R.S.T . . . 96
4.6 Mocking with Mockito . . . 97
4.6.1 Types of Test Double . . . 97
4.6.2 Activation . . . 98
4.6.3 Annotations . . . 99
4.6.3.1 `@Mock` . . . 99
4.6.3.2 `@Spy` . . . 100
4.6.3.3 `@Captor` . . . 100
4.7 Code Coverage . . . 101
4.8 Static Code Analysis . . . 103
4.9 Continuous Integration . . . 104

4.9.1 Differences between CI, CD, and CD . . . 105
4.9.2 CI Workflow . . . 107
4.9.3 Preconditions . . . 108
4.9.4 Advantages and Disadvantages . . . 109
4.9.5 Best Practices . . . 109

5. Design Principles . . . 110
5.1 Goal of Design Principles . . . 110
5.2 Overview of Design Principles . . . 110
5.3 SOLID Principles . . . 112
5.3.1 Single Responsibility Principle . . . 112
5.3.1.1 Example: `Modem` . . . 112
5.3.1.2 Example: `Book` . . . 115
5.3.1.3 Example: `Product` . . . 117
5.3.2 Open Closed Principle . . . 118
5.3.2.1 Example: `LoanRequestHandler` . . . 118
5.3.2.2 Example: `Shape` . . . 122
5.3.2.3 Example: `HumanResourceDepartment` . . . 124
5.3.2.4 Example: `Calculator` . . . 126
5.3.2.5 Example: `FileParser` . . . 128
5.3.3 Liskov Substitution Principle . . . 131
5.3.3.1 Example: `Rectangle` . . . 131
5.3.3.2 Example: `Coupon` . . . 135
5.3.3.3 Example: `Bird` . . . 139
5.3.4 Interface Segregation Principle . . . 142
5.3.4.1 Example: `MultiFunctionDevice` . . . 142
5.3.4.2 Example: `TechEmployee` . . . 145
5.3.4.3 Example: `StockOrder` . . . 149
5.3.5 Dependency Inversion Principle . . . 150
5.3.5.1 Example: `UserService` . . . 150
5.3.5.2 Example: `Logger` . . . 152
5.4 Packaging Principles - Cohesion . . . 154
5.4.1 Release Reuse Equivalency Principle . . . 154
5.4.2 Common Closure Principle . . . 154
5.4.3 Common Reuse Principle . . . 155
5.5 Packaging Principles - Coupling . . . 155
5.5.1 Acyclic Dependencies Principle . . . 155
5.5.1.1 Example: `Cyclic dependency` . . . 156
5.5.2 Stable Dependencies Principle . . . 160
5.5.3 Stable Abstractions Principles . . . 162
5.6 Further Design Principles . . . 165
5.6.1 Speaking Code Principle . . . 165
5.6.2 Keep It Simple (and) Stupid! . . . 165
5.6.3 Don’t Repeat Yourself / Once and Only Once . . . 166
5.6.4 You Ain’t Gonna Need It! . . . 166
5.6.5 Separation Of Concerns . . . 167

6. Design Patterns of the Gang of Four . . . 168
6.1 Creational . . . 171
6.1.1 Singleton . . . 171
6.1.1.1 Example: `Lazy loading` . . . 171
6.1.1.2 Example: `Eager loading` . . . 172
6.1.1.3 Example: `Enum singleton` . . . 172
6.1.2 Builder . . . 173
6.1.2.1 Example: `MealBuilder` . . . 174
6.1.2.2 Example: `PizzaBuilder` . . . 176
6.1.2.3 Example: `Email` . . . 178
6.1.2.4 Example: `ImmutablePerson` . . . 180
6.1.3 Factory Method . . . 183
6.1.3.1 Example: `Logger` . . . 183
6.1.3.2 Example: `Department` . . . 185
6.1.4 Abstract Factory . . . 188
6.1.4.1 Example: `Car` . . . 189
6.1.5 Prototype . . . 194
6.1.5.1 Example: `Person - Shallow copy` . . . 194
6.1.5.2 Example: `Person - Deep copy` . . . 197
6.1.5.3 Example: `Person - Copy constructor / factory` . . . 199
6.2 Structural . . . 202
6.2.1 Facade . . . 202
6.2.1.1 Example: `Travel` . . . 203
6.2.1.2 Example: `SmartHome` . . . 203
6.2.2 Decorator . . . 206
6.2.2.1 Example: `Message` . . . 207
6.2.2.2 Example: `Window` . . . 208
6.2.3 Adapter . . . 211
6.2.3.1 Example: `Sorter` . . . 212
6.2.3.2 Example: `TextFormatter` . . . 213
6.2.4 Composite . . . 215
6.2.4.1 Example: `Graphic` . . . 216
6.2.4.2 Example: `Organization Chart` . . . 218
6.2.5 Bridge . . . 220
6.2.5.1 Example: `Message` . . . 220
6.2.5.2 Example: `Television` . . . 223
6.2.6 Flyweight . . . 226
6.2.6.1 Example: `Font` . . . 227
6.2.6.2 Example: `City` . . . 228
6.2.7 Proxy . . . 230
6.2.7.1 Example: `Spaceship` . . . 230
6.2.7.2 Example: `ImageViewer` . . . 232
6.3 Behavioural . . . 234
6.3.1 State . . . 234
6.3.1.1 Example: `MP3Player` . . . 234
6.3.1.2 Example: `Door` . . . 237

6.3.2 Template Method . . . 240
6.3.2.1 Example: `Compiler` . . . 240
6.3.2.2 Example: `Callbackable` . . . 241
6.3.3 Strategy . . . 245
6.3.3.1 Example: `Compression` . . . 245
6.3.3.2 Example: `LogFormatter` . . . 247
6.3.4 Observer . . . 250
6.3.4.1 Example: `DataStore` . . . 251
6.3.4.2 Example: `Influencer` . . . 252
6.3.5 Chain of Responsibility . . . 254
6.3.5.1 Example: `Purchase` . . . 254
6.3.5.2 Example: `Authentication` . . . 257
6.3.6 Command . . . 260
6.3.6.1 Example: `FileSystem` . . . 260
6.3.6.2 Example: `Television` . . . 263
6.3.7 Interpreter . . . 266
6.3.7.1 Example: `HexBinary` . . . 266
6.3.7.2 Example: `Calculator` . . . 269
6.3.8 Iterator . . . 272
6.3.8.1 Example: `Cars - intern` . . . 273
6.3.8.2 Example: `Cars - extern` . . . 274
6.3.9 Mediator . . . 276
6.3.9.1 Example: `Chat` . . . 276
6.3.9.2 Example: `Aircraft` . . . 279
6.3.10 Memento . . . 282
6.3.10.1 Example: `Editor` . . . 282
6.3.10.2 Example: `Balance` . . . 286
6.3.11 Visitor . . . 288
6.3.11.1 Example: `Fridge` . . . 288
6.3.11.2 Example: `Figures` . . . 291

Bad code is like a joke, if you have to explain it, it is bad.

Preface

This is a *Forever Edition.* That means that this book will see periodic updates.

If you purchased this book, thank you very much! Writing a book like this takes some hours of effort. Please treat your book as your copy and do not redistribute the book without authorization.

If you find this book useful, I would greatly appreciate you purchasing a copy. By doing so, you'll be letting me know that books like this are useful to you.

What is the goal of this book?

Readers of this book will acquire in-depth knowledge and skills for the analysis, assessment, and improvement of software quality. You will be able to apply principles, patterns, techniques, and tools needed to write clean code.

Understanding important concepts of software quality in Java projects:

- Best Practices
- Functional and technological scope
- Development and programming models
- Tools

Learning Objectives:

- Acquisition of knowledge and skills to analyze, assess and improve software quality
- Learning principles, patterns, techniques, and tools
- Deeping technology knowledge

Which topics can you expect?

Software testing is not limited to a specific phase of a project. Already during the coding phase or the build process, critical and difficult to find software defects can be detected in the source code. The necessary procedures and tools are presented in this book.

Among other things, the following topics will be covered:

- Overview of the basics of software quality
- Software metrics, metric application in practice
- Structured design, cohesion, and coupling
- Overview of Principles, Best Practices, and Code Smells

- Compliance and verification of Java code conventions
- Static software testing, especially review techniques and static program analysis
- Ensure software quality with tools such as SonarQube, PMD, SpotBugs, Checkstyle, ArchUnit and Dependency-Track
- Software tests with JUnit and Mockito
- Checking the test code coverage
- CI/CD
- Design Principles
- Design Patterns (GoF)

Who should read this book?

I wrote the book because I want to help developers get experience in clean code or improve it. Get them a feeling of what it is about to be a Software Craftsman. It is about Professionalism, Pragmatism, and Pride!

It's a mindset, where software developers choose to be responsible for their careers, learning new tools and techniques, and constantly getting better. The journey to Software Craftsmanship is a long one.

That's why I wanted to share some techniques and practices for improving code quality and motivating developers.

What you must learn about Software Development yourself?

- **Developers are always wrong** - only the degree to which different developers are wrong in their endless architecture discussions differs!
- **If something can break, it will break** - make sure that a project is foolproof!
- **All code is bad** - there are only gradual gradations of the defectiveness of a piece of software!
- **There is always a bug somewhere** - you just must search long enough!
- **The most important thing is the customer** - and they don't give a damn about the technologies and processes used!
- **Project development on paper does not work** - problems are only identified during the development process!
- **Less is more** - if something is not necessary, just leave it out!
- **Only 20% of our work consists of coding** - the rest is designing, debugging, testing, meetings, discussing, etc.
- **The customer never knows what he wants** - everyone has only a vague idea of the desired result!
- **Someone has done it before** - don't reinvent the wheel!

What about the code examples and typographic conventions?

All the code examples in this book are in Java. I don‘t want to put much boilerplate code within the code examples, so I just leave it out.

Class names, data types, method and function names, and code are not included in `proportional font`.

Which literature is this book based on?

Schneider, **Kurt**: *Abenteuer Software Qualität – Grundlagen und Verfahren für Qualitätssicherung und Qualitätsmanagement*, dpunkt.verlag, 2007

Robert, **Martin**: *Clean Code – A Handbook of Agile Software Craftsmanship*, Prentice Hall, 2009

Lilienthal, **Carola**: *Sustainable Software Architecture. Analyze and Reduce Technical Debt*, Dpunkt Verlag, 2019

Bloch, **Joshua**: *Effective Java – Third Edition*, Addison-Wesley, 2017

Roock, **Stefan**: *Refactorings in grossen Softwareprojekten*, Dpunkt Verlag, 2004

Gamma, **Erich**: *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley Professional, 1994

Robert C. Martin: *Agile Software Development: Principles, Patterns and Practices*, Prentice Hall, 2003

Bugayenko, **Yegor**: *Elegant Objects Volume 1*, CreateSpace Independent Publishing Platform, 2016

Bugayenko, **Yegor**: *Elegant Objects Volume 2*, CreateSpace Independent Publishing Platform, 2017

Harrer, **Simon**: *Java by Comparison: Become a Java Craftsman in 70 Examples*, O‘Reilly UK Ltd., 2018

Robert, **Martin**: *Clean Architecture: A Craftman‘s Guide to Software Structure and Design*, Prentice Hall, 2017

Robert, **Martin**: *The Clean Coder: A Code of Conduct for Professional Programmers*, Prentice Hall, 2011

David, **Thomas**: *Pragmatic Programmer special 2nd*, Addison-Wesley Professional, 2019

Kaczanowski, **Tomek**: *Practical Unit Testing with TestNg*, 2012

Meszaros, **Gerard**: *xUnit Test Patterns*, Addison-Wesley, 2007

Long, **Fred**: *Java Coding Guidelines: 75 Recommendations for Reliable and Secure Programs*, Addison-Wesley Professional, 2013

Sandro, **Mancuso**: *The Software Craftsman: Professionalism, Pragmatism, Pride*, Pearson, 2014

Freeman, **Steve**: *Growing Object-Oriented Software, Guided by Tests*, Addison-Wesley, 2009

Giving feedback?

If you have anything to say about this book, I'd love to receive your feedback.

Found a typo? Discovered a code bug? Have a content suggestion? Let me know and be a part of this book by providing feedback.

For contributions, please always tell me the chapter name, heading reference, and a brief snippet of text as a reference.

You can use the Leanpub forum of this book to post errata and ask questions about this book.

Since this book is self-published, I will create as many releases as I wish.

Thank you for your time to help make this book better!

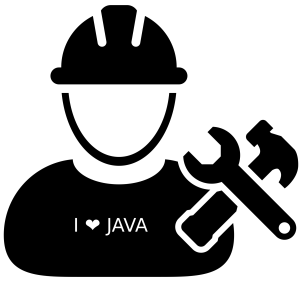
You can provide feedback under this e-mail:
clean-code-fundamentals@gmail.com
or via Twitter:
#CleanCodeFundamentals

1. Introduction to Software Craftsmanship and Clean Code

1.1 A Passion for Software Development

Software Craftsmanship is a long journey to mastery. It's a mindset where software developers choose to be responsible for their own careers, constantly learning new tools and techniques and constantly bettering themselves.
–Sandro Mancuso, The Software Craftsman

The basis of the Software Craftsmanship movement is the publication of *Clean Code* by Robert C. Martin. The author is a co-founder of the Software Craftsmanship movement. There are many initiatives like the *Manifesto for Software Craftsmanship* or the *Clean Code Developer* movement which raised the bar regarding software development - luckily.



Software Craftsmanship

Software Craftsmanship focuses on the adoption of good technical practices in the development process. It is obvious, that everyone likes Software Craftsmanship. Everybody wants to have a clean code base, but it's difficult to achieve. Several steps should also be done in the right way. The practices defined under the Software Craftsmanship movement are the ones that will help you to achieve this.

Software Craftsmanship should be the goal you are aiming for. It is not only the maturity of your project, it is the quality of your software and your codebase as well. As Benjamin Franklin said: *The bitterness of poor quality remains long after the sweetness of low price is forgotten.*

Some developers are enthusiastic about their work and have already acquired these values to provide quality and kindness to the code. They have the necessary skills to realize products according to their values. Unfortunately, many others know little or not at all that there is another way to develop software. Even today, many development practices that help to create well-crafted software are simply not taught in schools.

If you work in an environment where these practices simply are not applied or there is no room for well-crafted software, ask yourself the question if you can change this. Be a craftsman, and spread the professionalism of software development out to the team with the hope that individuals will follow you. Adopt a culture where the values of Software Craftsmanship have the appropriate value. Do not rank other developers, maybe they had never the chance to work in this way or had never the luck having a good mentor. When you have success with this approach, fine. If not, you may be in the wrong company for you!

To become a software craftsman, you must first apprentice under a more experienced software craftsman. This will allow you to learn the trade and develop the skills needed to become a software craftsman yourself.

No one becomes a software craftsman overnight. It takes years of study and practice to master the techniques and principles. Good software craftsmen are always learning and constantly striving to be better.

Practices that help you strengthen your passion as a software developer:

Understand the importance of *good enough*.

A working good enough solution is usually better than a non-existent perfect solution. Learn to balance constraints. Don't get bogged down in the war of beliefs about technologies and the right solution, be pragmatic and effective in what you do. Know where and when to compromise and when to say *enough*.

If you get stuck, don't hesitate to ask for help.

Don't spend your time endlessly trying to solve a problem that can easily be solved by someone else. There are people on your team that can you help to solve your problem much faster and give you valuable tips. Know when to keep trying and when to stop and ask for help.

If you are not able to communicate your ideas and connect with others, you will not get far.

Learn how to communicate effectively at every level with the people around you. Get involved with other ideas, listen to them, and expand your horizons with other points of view. This will help you to question your views and, in the end, to communicate them to others.

You will be criticized. Learn how to deal with it.

Understand the value of criticism and learn how to make the most of it. These are opportunities for you to grow and should be valuable feedback provided for you.

If you want to make a difference, you need to be effective in prioritizing.

Not everything you can do is always worth doing. Understand how to identify high-impact work and dismiss the rest. Understand the basics and concepts and deepen them. Focus on a few things that you are very interested in while keeping track of other things. Nowadays frameworks come and go, it is more important to understand the concepts behind them. You can apply these to all frameworks coming.

Learn how you be effective in learning.

Find out the best way for you to learn new things over time. Explaining things to others and sharing your knowledge is a very effective way to deepen your knowledge and learn new things.

Be selective with your valuable time.

You cannot do it all by yourself. If you learn how to delegate effectively, you can multiply your time for topics that provide more value to you. Be realistic with your goals, you can't know everything.

Share all you learn with everyone else.

Not only you will help others around you to become better, but you will also be strengthening your understanding and growing your knowledge. When knowledge grows in a team and exchange takes place, everyone wins.

Learn how to motivate others.

If you know how to light a fire and keep it alive, everyone will want to stay around you. Allow your team members to work in their way and give them the freedom they need to work creatively and effectively. Set a good example and find out what your colleagues need to be motivated.

Learn to be proactive.

Don't wait for problems to become problems. Take care of things in a proactive way.

Embrace change.

Change is a natural part of life, and it should be embraced rather than feared. The world is constantly evolving, and software needs to change with it. To stay relevant, software must be constantly updated and improved. Change can be difficult, and it can be scary. But it is also essential for growth. Embracing change is the only way to ensure that your software will evolve and continue to be useful for years to come.

Refactor fearlessly.

To achieve software craftsmanship, one must be willing to refactor fearlessly. This means being willing to change the structure of your code to make it more readable, maintainable, and extensible. It also means being willing to throw away code that is no longer needed. Refactoring can be scary, especially when you're working on code that is already working. But remember, the goal is not to avoid breaking things but to make your code better. And if you're not afraid to break things, you're more likely to end up with truly great code.

1.2 Manifesto for Software Craftsmanship

The famous Manifesto for Software Craftsmanship[1] describes professional software developers, and itself reads like an addition to the Agile Manifesto[2]:

In the Manifesto for Software Craftsmanship, the authors make a case for why software development should be considered a craft. They argue that, like any other craft, software development should be done with an emphasis on quality, and that developers should take pride in their work. They also believe that developers should be lifelong learners, always looking to improve their skills.

While the Manifesto for Software Craftsmanship is focused on software development, many of its principles can be applied to other fields as well. In any field, quality should be a primary concern, and pride should be taken in one's work. Additionally, lifelong learning is essential in any field to stay ahead of the curve and remain competitive.

As aspiring Software Craftsmen, we are raising the bar of professional software development by practicing it and helping others learn the craft. Through this work we have come to value:

- Not only working software, but also **well-crafted software**
- Not only responding to change, but also **steadily adding value**
- Not only individuals and interactions, but also **a community of professionals**
- Not only customer collaboration, but also **productive partnerships**

That is, in pursuit of the items on the left we have found the items on the right to be indispensable.

These principles are important because they offer a roadmap for creating software that is not only functional but also well-designed and maintainable. Following these principles will help software developers to create code that is more likely to be used and maintained over the long term.

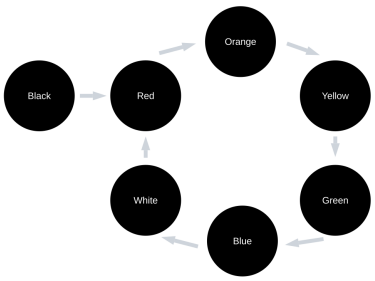
[1] http://manifesto.softwarecraftsmanship.org/
[2] http://agilemanifesto.org/

1.3 Clean Code Developer

The Clean Code Developer (CCD)[3] is an initiative for more professionalism in software development. CCD defines a collection of principles and practices. These are divided into different degrees. However, it also includes a collection of specific values, such as **evolvability**, **correctness**, **production efficiency** and **continuous improvement**.

Through constant iteration of the different degrees, these principles and practices are becoming part of the way we act. This creates an awareness of quality within the development process. Those who practice daily will be able to intuitively retrieve this knowledge

Below you can see an overview of the degrees of the CCD system, which the iterative cycle goes through. The degrees build on each other in terms of difficulty and benefits. A developer who has not yet started with CCD owns the black grade. If someone has not yet started with CCD own the black grade. A black grade only indicates the initial interest in CCD.



Clean Code Developer Grade Cycle

Clean Code Developer Grades

Grade	Principles	Practices
Red	Don't Repeat Yourself (DRY)	Using a Version Control System
	Keep it simple, stupid (KISS)	Refactoring Patterns Rename and Extract Method
	Beware of Optimizations	Boy Scout Rule
	Favour composition over inheritance	Root Cause Analysis
		Daily Reflection
Orange	One level of abstraction	Issue Tracking
	Single Responsibility Principle (SRP)	Automatic Integration Tests
	Separation of Concerns (SoC)	Read, Read, Read
	Source Code Convention	Reviews
Yellow	Information hiding	Attend Conferences

[3] https://clean-code-developer.com/

Clean Code Developer Grades

Grade	Principles	Practices
	Principle of least astonishment (POLA)	Automatic Unit Tests
	Liskov Substitution Principle (LSP)	Mockups
	Interface Segregation Principle (ISP)	Code Coverage Analysis
	Dependency Inversion Principle (DIP)	Complex Refactoring
Green	Open Closed Principle (OCP)	Continuous Integration
	Tell, don´t ask	Static Code Analysis
	Law of Demeter (LoD)	Inversion of Control Container
		Share your Experience
Blue	Implementation matches Design	Continuous Deployment
	You Ain't Gonna Need It (YAGNI)	Component Orientation
	Separation of Design and Implementation	Iterative Development
		TDD
White	All principles flow together	All practices flow together
	and includes all other grades	and includes all other grades

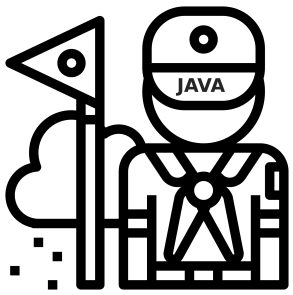
A developer finishes with the white grade when he is aware of all the CCD values. In this grade, he follows a constant repetition of the cycle, which serves the purpose of refining the use of the aspects of the individual grades. This reflects our job as developers, in which we constantly keep our knowledge up to date and refine our tooling and practices. Only experienced and advanced software developers will be able to work in white grade.

A more visualized overview of all principles, practices, and values can be found here.[4][5]

[4] https://unclassified.software/de/topics/clean-code
[5] http://michael.hoennig.de/download/CCD-Poster.pdf

1.4 Boy Scout Rule

Always leave the campground cleaner than you found it.
–Robert C. Martin[6]



Boy Scout Rule

Boy Scouts have a rule regarding camping that they should leave the campground cleaner than they found it. By ensuring this rule, they can guarantee that they do not cause new problems, at least as far as the cleanliness of the area is concerned.

What if we apply this rule to our code?

The Boy Scout Rule suggests that you simply try to make sure that each time you commit, you leave the code better than when you found it. This means that every time you make a change, you improve the codebase. Maybe only slightly. Improve a variable name, clean up incomplete documentation, extract a large method to multiple smaller ones or add a missing test.

Keeping code clean is a constant challenge, and developers and software teams must decide if, when, and how they want to keep their code clean. Resolving these technical debts[7] through refactoring is necessary to keep the code in a state where it remains economically viable to extend and to maintain it.

When does it make sense to spend time working on improving the code?

Over time, the quality of the source code on which a system is based tends to deteriorate, leading to increasing technical debt. Some teams take the approach of discontinuing all value-adding work and trying to clean up the codebase. This usually results in some refactoring sprints or even a complete rewrite. Neither of these is a solution from a business perspective. Feature development and bug-fixing must continue, and refactoring should not take up most of the time. According to the Boy Scout Rule, teams can improve the quality of their code over time while continuing to deliver value to their customers and stakeholders. In this way, continuous improvement allows you to move in a corridor where technical debt does not dominate your system. Do not let your system exceed a point where the application becomes unmaintainable. Thus "*Try to leave the world a little better than you found it*"[8].

[6] http://cleancoder.com/
[7] WardCunningham:*TheWyCashPortfolioManagementSystem*;ExperienceReport,OOPSLA'92,1992
[8] Robert Baden-Powell, the father of the Boy Scouts, and the original of this expression. https://en.wikipedia.org/wiki/Robert_Baden-Powell,_1st_Baron_Baden-Powell

1.5 Broken Windows Theory

The importance of fixing the small problems in your code, the "broken windows", so they don't grow into large problems.
–The Pragmatic Programmer[9]

The broken windows theory is a criminological theory that states that visible signs of crime, anti-social behavior, and civil disorder create an urban environment that encourages further crime and disorder, including serious crimes. The theory suggests that policing methods that target minor crimes, such as vandalism, loitering, public drinking, jaywalking and fare evasion, help to create an atmosphere of order and lawfulness, thereby preventing more serious crimes. –*Wikipedia*[10]

Broken Windows Theory

The Broken Window Theory can be used as a metaphor for software development and technical debt on a project.

Every time you delay a fix for a known problem, you create debt. You might know that something is broken, but if you don't fix it now, you're in debt. If you have some of these and you are aware of them, that is fine. You make a notice and remove them later. But just like with real debt, it doesn't take much to get to a point where you can never pay it back. It is the point where you have so many problems that you can never go back and solve them.

You do not want to let technical debt get out of hand. Stop the small problems before they grow into bigger ones. Don't allow broken windows in your project, but fix them. If you can't fix them now, create a technical debt ticket to make sure everyone knows they're broken. A technical debt log helps to provide an overview within a team.

As soon as something is broken and cannot be repaired, it starts to spread throughout the team. The team or new team members get used to these Broken Windows and are not afraid to add new ones. This leads to even worse code and the loss of respect for how you handle the code. Take care of technical debts and solve problems no matter how small they are. Don't let the code become rotten. The better the code, the less other team members will be tempted to check in bad code.

[9] https://pragprog.com/titles/tpp20/the-pragmatic-programmer-20th-anniversary-edition/
[10] https://en.wikipedia.org/wiki/Broken_windows_theory

1.6 Cargo Cult Programming

Cargo Cult

In the software development world, there is a phenomenon known as Cargo Cult Programming[11]. The term was first coined by computer scientist Richard P. Gabriel in his 1996 essay *Cargo Cult Software Engineering*. Gabriel used the term to describe the phenomenon of teams of programmers who blindly follow development methodologies without understanding why they are doing so.

Gabriel compared Cargo Cult Programming to the religious Cargo Cults of Melanesia. These cults sprang up in the aftermath of World War II as native Melanesians witnessed the arrival of American soldiers with their cargo of supplies. The Melanesians built replicas of airstrips and control towers in an attempt to attract more American cargo to their island. Similarly, cargo cult programmers will replicate the development practices in the hope of achieving the same results.

It is characterized by the ritual of code or program structures that serve **no real purpose**. The term Cargo Cult Programmer may apply,
when an **unskilled** or **novice computer programmer** copies some program code from one place to another with little or **no understanding** of how it works or whether it is required in its new position. It can also refer to the results of applying a design pattern or coding style blindly **without understanding** the reasons behind that design principle.

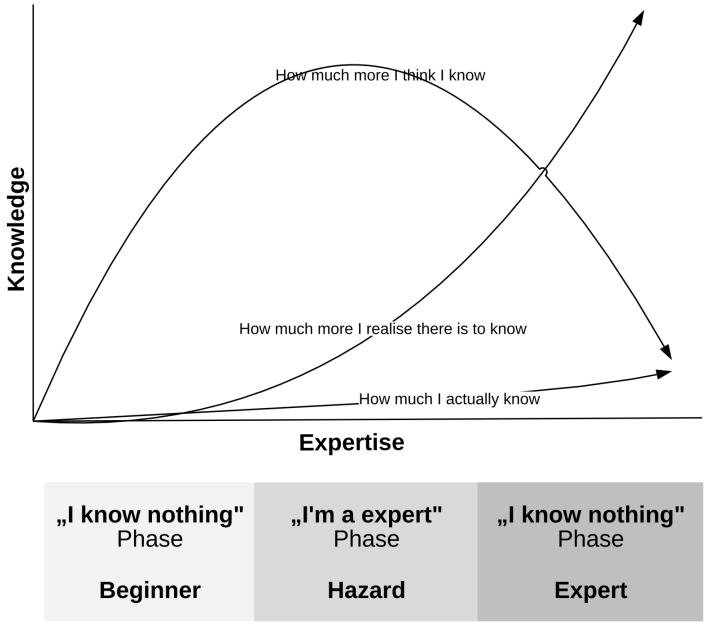
Cargo Cult Programming is a dangerous form of programming that can lead to buggy, unstable code. To avoid Cargo Cult Programming, programmers need to be aware of its dangers and be vigilant in their code reviews.

Overall, Cargo Cult Programming is a harmful development practice that can lead to disastrous consequences for software projects. Development teams need to be aware of the dangers of this programming technique and take steps to avoid it.

[11] https://archive.org/details/professionalsoft00mcco_0

1.7 Knowledge - Expertise

Simon Wardley published this graph as a joke[12] which it's not based on any research. Nevertheless, you can recognize yourself as a developer in this graphic. The more you study a programming language or technology, the more you think you know about it. But at some point, you have to admit to yourself that you know nothing at all. But we shouldn't despair, we should remain curious and should constantly learn.

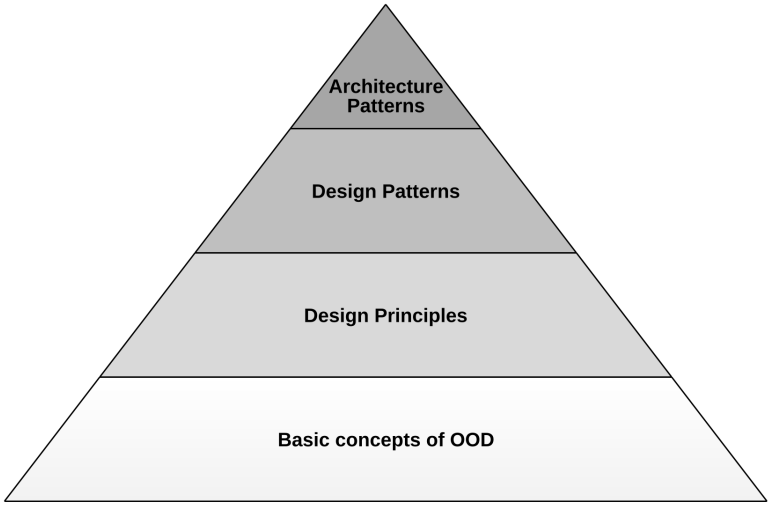


Knowledge - Expertise

[12] https://threadreaderapp.com/thread/1045567546034987008.html

2. Basics of Software Design

2.1 Software Design Pyramid



Software Design Pyramid

Basic concepts of OOD

The basic concepts of OOD are the prerequisite for this book and will be discussed only briefly in the following chapter. However, they are the basis for understanding the design principles and design patterns in the later chapters. The language specification and features of Java and the implementation of polymorphism, encapsulation, abstraction, and inheritance help to understand and implement them.

Design Principles

The Design Principles are based on the concepts of the OOD. The best known are the SOLID principles of Robert C. Martin. SOLID is an acronym for five Design Principles intended to make software designs more understandable, flexible, and maintainable. These are explained in detail in this book. Besides these, there are many others, which are also described and explained in this book.

Design Patterns

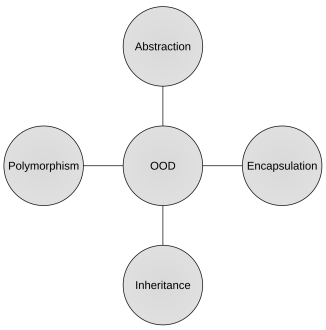
The Design Patterns go one step further and offer solution patterns for recurring problems. Probably the best known are the GoF Design Patterns, which are covered in this book with many examples. Besides the GoF Patterns, there are many more which are also worth studying.

Architecture Patterns

The Architectural Patterns are at the top of the design pyramid. Architectural Patterns are methods of proven good design structures. More specifically, an Architectural Pattern is a set of design decisions that are repeated in practice, and have well-defined characteristics which can be reused. There are many Architecture Patterns like microkernel, micro-services, layered architecture, event-based, and many more.

2.2 Basic concepts of OOD

Object-oriented programming has four basic concepts: **abstraction**, **encapsulation**, **inheritance** and **polymorphism**. Although these concepts may seem complex, understanding how they work will help you to understand the basics of software design.



Basic concepts of OOD

Abstraction

Abstraction implies that the client only interacts with specific attributes and methods of an object. Abstraction uses simplified high-level access to a complex object.

- Hiding of complexity by ignoring irrelevant details
- Solves the problem at the design level

Encapsulation

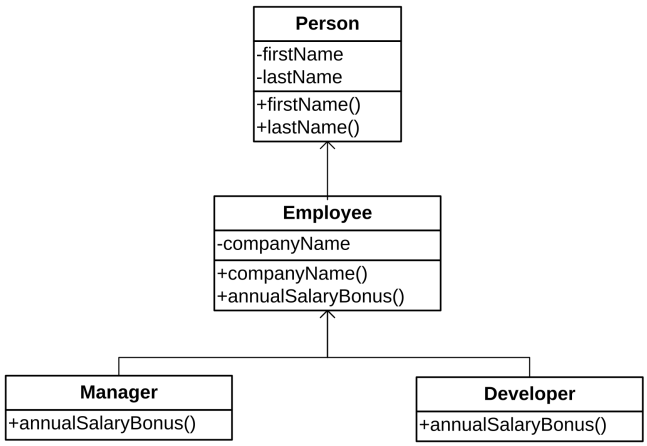
Encapsulation is the mechanism for hiding data implementation by restricting access to classes, methods, and variables. An example would be achieving this, by keeping instance variables private and making access methods more visible. To reduce couplings between software components, encapsulation mechanisms are essential.

- Information and data hiding
- Solves the problem at the implementation level

Inheritance

Inheritance allows classes to inherit characteristics of other classes. In practice, parent classes extend attributes and behaviors to subclasses.

- Extension and specialization
- Inheritance supports reusability

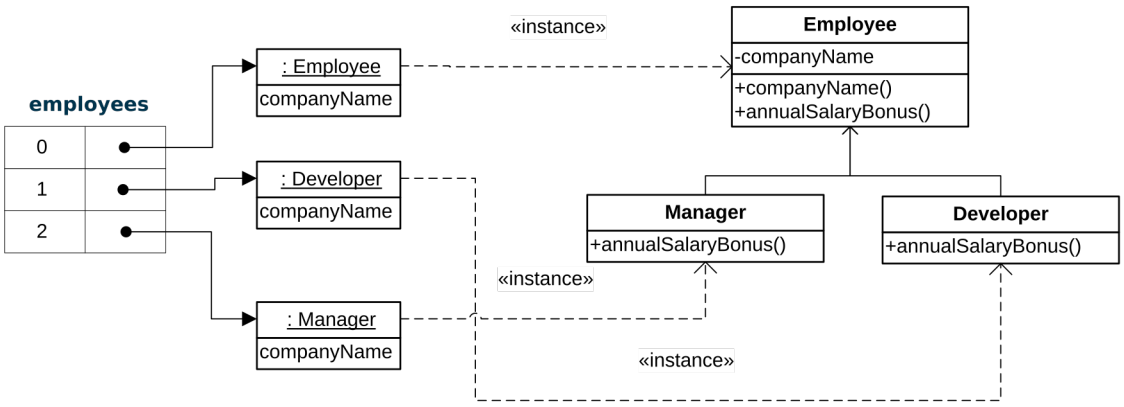


Inheritance

Polymorphism

Polymorphism means one name but many forms. It allows designing objects that share behaviors. Using inheritance, objects can overwrite parent behavior with specific behavior. Polymorphism allows the same method to perform different behavior statically and dynamically. The static one is achieved by method overloading and the dynamic by method overriding. Thus, polymorphism is closely connected with inheritance. We can write code that works on the superclass, and it works with any subclass type as well.

- Substitutability
- Same interface, different behavior



Polymorphism

Polymorphism

```
class Client {

    public static void main(String[] args) {
        List<Employee> employees = List.of(
                new Employee("Employee", "Max", "Company"),
                new Developer("Developer", "Max", "Company"),
                new Manager("Manager", "Max", "Company"));

        employees.forEach(e -> System.out.printf(assembleMessage(e)));
    }

    private static String assembleMessage(Employee employee) {
        return String.format("%s with an annual salary bonus of %g!\n",
                employee.getClass().getName(),
                employee.annualSalaryBonus());
    }
}
```

Output

```
Employee with an annual salary bonus of 1000,00!
Developer with an annual salary bonus of 3000,00!
Manager with an annual salary bonus of 5000,00!
```

In this example, the compiler does not know what type of the reference attribute `employee` is. They do not know which operation is called on `annualSalaryBonus()`. The method of the superclass `Employee` or the redefined `annualSalaryBonus()` of the subclass `Developer` or `Manager`.

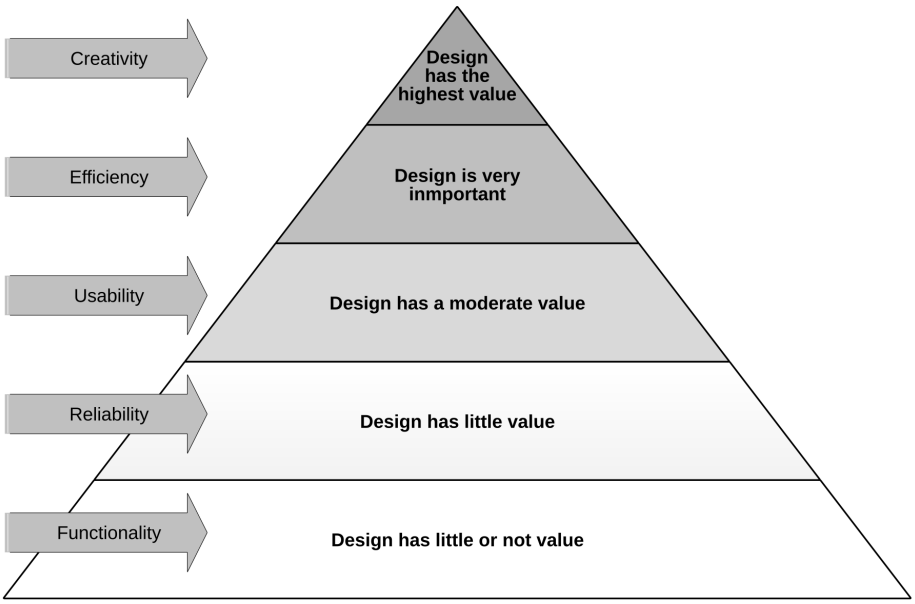
Furthermore, the operation `assembleMessage()` does not need to be modified if additional subclasses are added to the `Employee` superclass.

2.3 Goals of Software Design

Software design is invisible: software that meets all functional requirements can still be poorly designed.

- **Design is good when**
  - it breaks down the complexity of the software into manageable and simple problems.
  - small interfaces were defined.
  - the components are decoupled.
  - the components have clearly defined responsibilities.
  - the software is maintainable.
  - the software can be easily changed and extended.
  - the software is stable.
  - Bugs can be fixed quickly.
  - the software is reusable in other software projects.
  - the code is understandable.

The figure below shows the value that software design can have in a project.



Design hierarchy of needs

Unfortunately, software design has not always been a high priority in a project. This can be due to various reasons, a lack of technical experience or simply the fact that it is considered to be

unimportant. From my point of view, it is a fatal mistake if a solid foundation is not created at the beginning of a project. It is almost like trying to build a house. You cannot build a house without a good foundation, and you start building from there.

2.4 Symptoms of *bad* design

Many signs can indicate a bad design in architecture. Some of these indicators are more obvious than others, but all of them can lead to problems down the road if they are not addressed.

One of the most common indicators of a bad design is a high degree of coupling between components. Coupling is the degree to which one component depends on another. When components are highly coupled, they are more difficult to change without affecting the other component. This can lead to a lot of problems when trying to make changes to the system later on.

Another common indicator of a bad design is a lack of modularity. Modularity is the degree to which a system is composed of separate, independent modules. A good design will have a high degree of modularity, which makes it easier to add or remove functionality without affecting the rest of the system. A system with a low degree of modularity is more difficult to change and can be more fragile as a result.

There are many other common causes of bad design, but some of the most common include a failure to understand the problem, a lack of foresight, and a lack of communication among team members. Often, bad design is the result of a rushed or poorly planned project.

Some common causes of bad design:

- **High coupling between components**
- **Lack of modularity**
- **Never-touch-running-code Syndrome**
  - Developers are afraid to change code.
  - Many workarounds, code is developed around it.
  - Changes have unknown and undetected side effects.
- **Small change in requirements leads to big changes in code**
- **Reuse through code duplication (Copy-Paste)**
  - Developers chase after the places that need to be changed.
  - The more code the more difficult it becomes to keep track of duplicates.
  - Errors have to be patched several times at different places.
- **Cyclic relations between artifacts**
  - Artifacts that are cyclically coupled cannot be tested individually.
  - Artifacts that are used in different cycles often play several roles, which makes them difficult to understand.
  - Artifacts that are used in different cycles cannot be exchanged easily.

2.5 Criteria for *good* design

When it comes to designing applications, there are a few key criteria that need to be met to create a successful, well-designed architecture.

Functionality is the most important aspect of any design. This means that the design should be able to perform the required tasks and functions effectively and efficiently.

Further, your architecture should be flexible and easily adaptable to changing needs and requirements. As your application changes and grows over time, your architecture should be able to accommodate those changes with minimal effort.

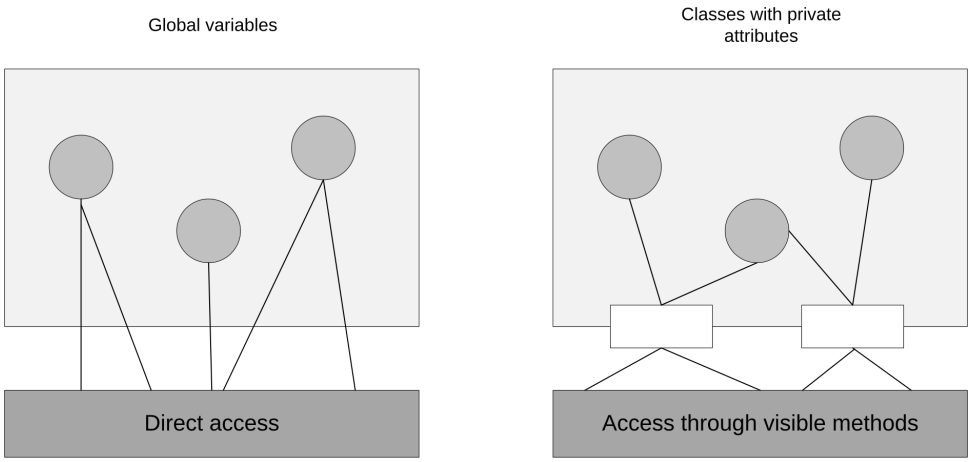
Furthermore, your architecture should be easy to maintain and understand. This is especially important if you have a team of developers working on the same code base. If your architecture is too complex or convoluted, it will be very difficult for developers to make changes or add new features without breaking something else. By keeping your architecture simple and straightforward, you can avoid a lot of headaches down the road. Finally, the design should be scalable. This means that it should be able to handle increased workloads without performance issues.

These criteria are fractal, i.e., they apply to all levels of the design like architecture, subsystems, and components.

- **Correctness**
  - Meeting all specified requirements
  - Ensuring the functionality of the system model
  - Adhering to non-functional requirements
- **Comprehensibility**
  - Creating self-explanatory code and design
  - Providing comprehensive documentation
- **Simplicity**
  - Striving for simplicity and straightforwardness
- **Flexibility and Adaptability**
  - Easy adaptation to changing needs
- **Modularity**
  - Promoting high cohesion for well-structured components
  - Minimizing coupling to reduce interdependencies
- **Reusability**
  - Designing for component reuse
- **Customizable**
  - Allowing for easy customization to meet specific user requirements
- **Dependency Management**
  - Avoiding cyclic dependencies for improved maintainability
- **Scalability**
  - Designing for potential growth and increased workload
- **Resilience**
  - Building systems capable of withstanding failures and recovering gracefully

2.6 Information Hiding

Simplify and reduce access to a class by hiding details, methods, and members that shouldn‘t be called and accessed by a client.



Information Hiding

In Java, classes and class members are access controlled. This mechanism prevents the users of a package or class from depending on unnecessary details of the implementation of that package or class.

The access is specified by the access modifiers `public`, `protected` or `private`. In the absence of an access modifier, it is the default access, also called package-private access.

The following table and illustration show the access control rules in Java.

Visible for:	public	protected	(default)	private
Same Class	yes	yes	yes	yes
Different Class, Same Package	yes	yes	yes	no
Different Class, Different Package	yes	no	no	no
Subclass, Same Package	yes	yes	yes	no
Subclass, Different Package	yes	yes	no	no

Visibility in Java

According to the Java Language Specification (JLS)[1]:

`public`

A `public` class member or constructor is accessible throughout the package where it is declared and from any other package.

[1] https://docs.oracle.com/javase/specs/jls/se15/html/jls-6.html#jls-6.6

`protected`

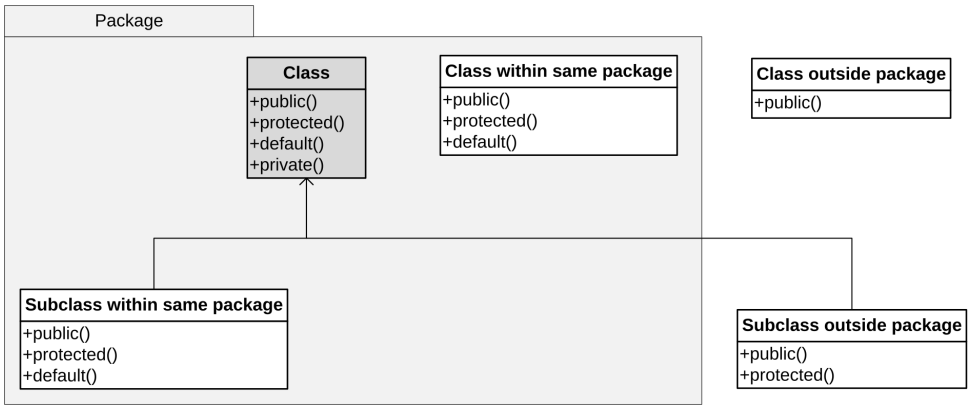
A `protected` member or constructor of an object may be accessed from outside the package in which it is declared only by code that is responsible for the implementation of that object. Can only be applied to fields, constructors, and methods, not to classes.

`package-private`

If none of the access modifiers `public`, `protected`, or `private` are specified, a class member or constructor has package-private access. It is accessible throughout the package that contains the declaration of the class in which the class member is declared, but the class member or constructor is not accessible in any other package.

`private`

A `private` class member or constructor is accessible only within the body of the top-level class that encloses the declaration of the member or constructor.



Visibility in Java

Declare members as `private` and provide accessible methods for `private` fields. Let distributed classes communicate only through this method calls.

If a class, interface, method, or field is part of a published API, it can be declared `public`. Other classes and members should be declared as either package-private or `private`. The exposure of fields and methods that give access to the mutable state of a class via interfaces must be avoided. This is because interfaces only allow publicly accessible methods, which are part of the API of the class. An exception is the implementation of methods that expose a public immutable view of a modifiable object. Modifiable classes should provide copy functionality to allow secure passing of instances to the client code.

Inheritance and visibility:

If a method is overwritten, the new method cannot have stronger access control than the original. In the case of `private` no inheritance is possible. More precisely, no overwriting takes place and access to the superclass method via `super` is not possible.

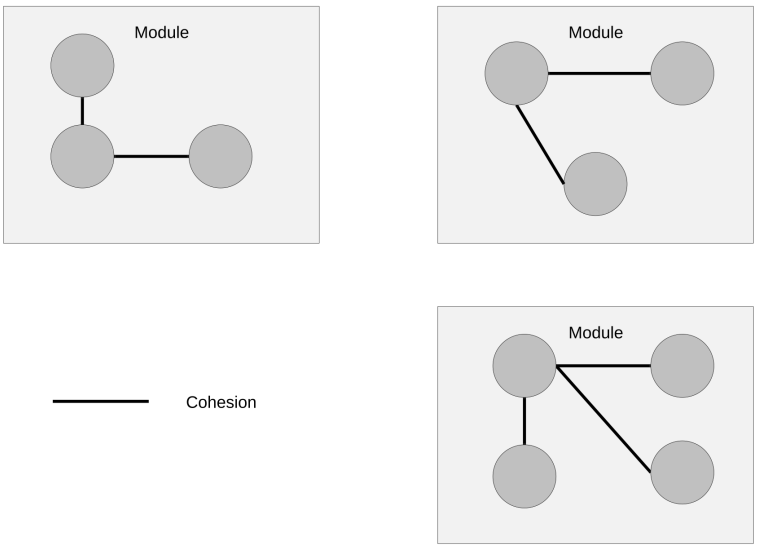
From / To:	public	protected	(default)	private
public	yes	no	no	no
protected	yes	yes	no	no
(default)	yes	yes	yes	no
private	no	no	no	no

Visibility and Inheritance

2.7 Cohesion

*Measure of the **affiliation** of the elements of a component.*

High cohesion of a component improves understanding, maintenance and adaptation.



Cohesion

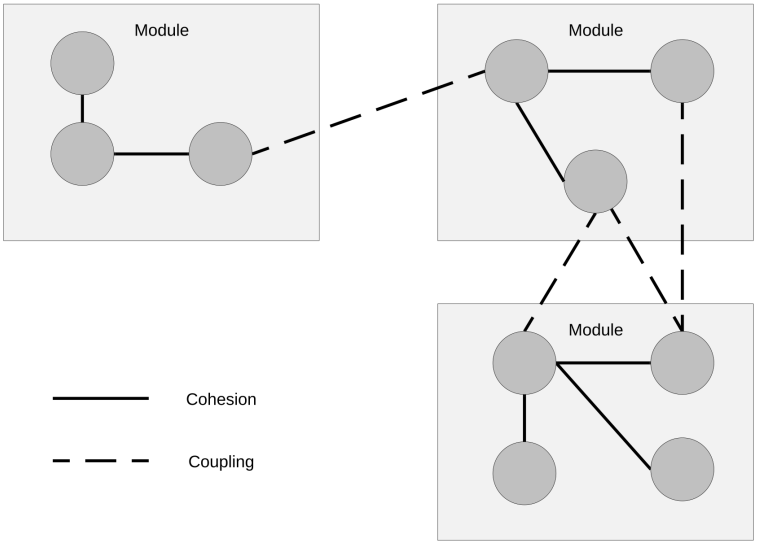
Easy to maintain code usually has high cohesion. The elements within the module are directly related to the functionality that the module is intended to provide. We can easily design, write and test our code because the code for a module is all together and works together.
Low cohesion would mean that the functionality is distributed throughout your codebase.

- **Degrees of cohesion**
  - functional cohesion
  - sequential cohesion
  - communicational cohesion
  - procedural cohesion
  - temporal cohesion
  - logical cohesion
  - coincidental cohesion

2.8 Coupling

*Measure for the **dependencies** between components.*

Low coupling facilitates maintainability and makes the system more stable.



Coupling

Modules should be as independent as possible from other ones. Changes to one module should not have a major impact on other modules. High coupling would mean that modules know too much about the internal of other application parts. Knowing too much about others makes changes difficult to coordinate and increases fragility.

With a low coupling, you can easily make changes to the internals without worrying about their effects on other modules. The low coupling also makes it easier to design, write and test code, as our modules are independent. We also have the advantage that our code is easy to reuse.

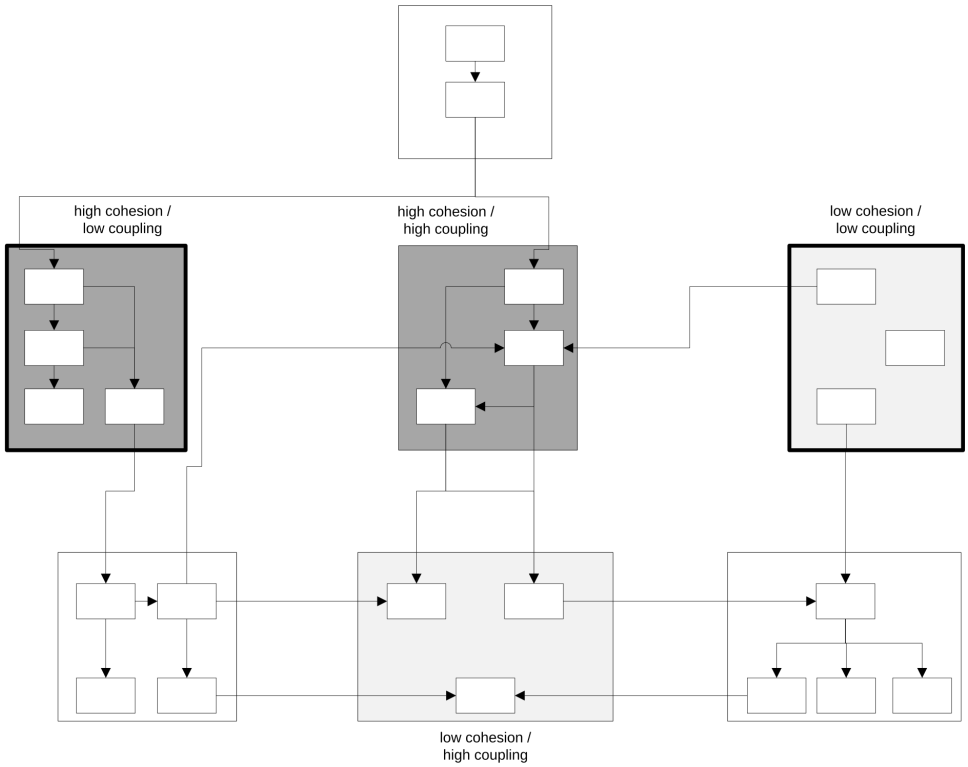
- **Consequences of high coupling**
  - Difficult to maintain, difficult to adapt
  - Monolithic
  - Exchange of components hardly possible
  - System becomes fragile
  - Difficult to reuse individual components
- **Types of coupling**
  - Data coupling
  - Interface coupling
  - Structural coupling
- **Reduction of coupling**
  - Coupling can never be reduced to zero!
  - Higher interface coupling increases flexibility
- **High cohesion enables low coupling**

2.9 Cohesion - Coupling

Cohesion and coupling are among the most important attributes for the quality of a design. In software development, coupling means the measure for the dependencies between components, while cohesion stands for the measure of the unity of the elements of a component. It is very important to understand what impact these two parameters can have on a system and how they can be influenced.

To sum it up:

- Cohesion represents the degree to which a part of a codebase represents a logically single, atomic unit.
- Coupling represents the degree to which a single unit is independent of others.
- Encapsulate information, make modules **highly cohesive**, and **decrease coupling** among modules.
- Try to follow the guideline at all levels of your codebase.
- It is impossible to achieve complete decoupling without affecting cohesion, and vice versa.



Cohesion - Coupling

2.10 Big Ball of Mud

A Big Ball of Mud is a haphazardly structured, sprawling, sloppy, duct-tape-and-baling-wire, spaghetti-code jungle. These systems show unmistakable signs of unregulated growth and repeated expedient repair. Information is shared promiscuously among distant elements of the system, often to the point where nearly all the important information becomes global or duplicated.

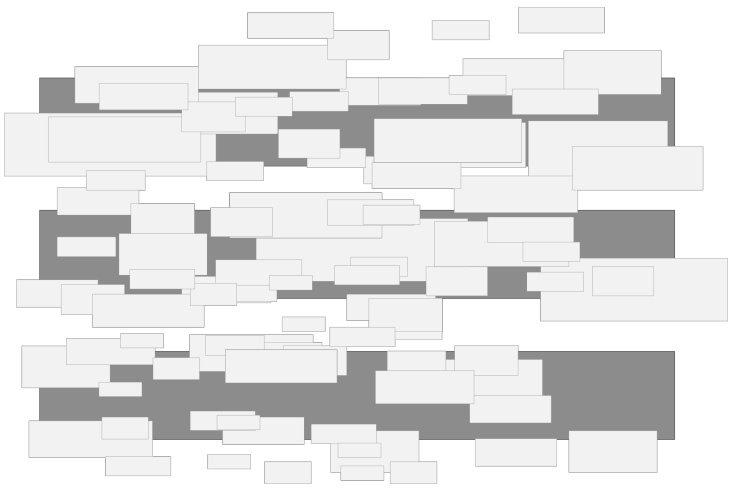
The overall structure of the system may never have been well defined.

If it was, it may have eroded beyond recognition. Programmers with a shred of architectural sensibility shun these quagmires. Only those who are unconcerned about architecture, and, perhaps, are comfortable with the inertia of the day-to-day chore of patching the holes in these failing dikes, are content to work on such systems.
–Brian Foote and Joseph Yoder, Big Ball of Mud[2].

Big Ball of Mud is an anti-pattern of software architecture and describes a program that has no recognizable software architecture.

Most frequent causes:

- Insufficient experience
- Lack of awareness for software architecture
- Not modularized, high coupling and low cohesion
- Does not comply with the principles of good design
- Pressure on the implementation team
- Constantly changing requirements
- High time pressure
- Low budget
- Employee fluctuation



Big Ball of Mud

[2] http://www.laputan.org/mud/mud.html#BigBallOfMud

2.11 Architecture Principles

Keep it simple stupid (KISS)

- Think of several options and then try the simplest option first.
- Always ask: *Is there an easier way to do this?*
- Think about the future maintainers, assume that they will be you, and work on that basis.
- When you look at existing systems, ask: *Do I need all this stuff* or *Do we need it at all?*

You're not going to need it (YAGNI)

- The architecture should be intended to support current and future requirements agreed with business stakeholders.

Principle of least astonishment (POLA)

- Keep things consistent so that people are not surprised when they find that a similar task is being done in a different place differently.
- If a difference is needed, document why.
- Consistency guides understanding, if you name things wrong or call the same thing by different names, you increase complexity.

Separation of concerns (SoC)

- Keep related things together, and unrelated things apart.

If it hurts, do it more often

- When we try to avoid difficulties, we are faced with increasing complexity. So, complexity is reduced if we perform difficult tasks more often.
- https://martinfowler.com/bliki/FrequencyReducesDifficulty.html

Leave your code better than you found it (Boy Scout Rule)

- Keep improving your code continuously, meaning that every time that we make a change in our code base, improve the code.
- No matter what the reasons are, we don't want to end up with additional technical debt. Every time we see code smell, we should try to eliminate the rot.
- If we all follow this approach, the system would gradually improve, and the deterioration would stop.

Talk to people

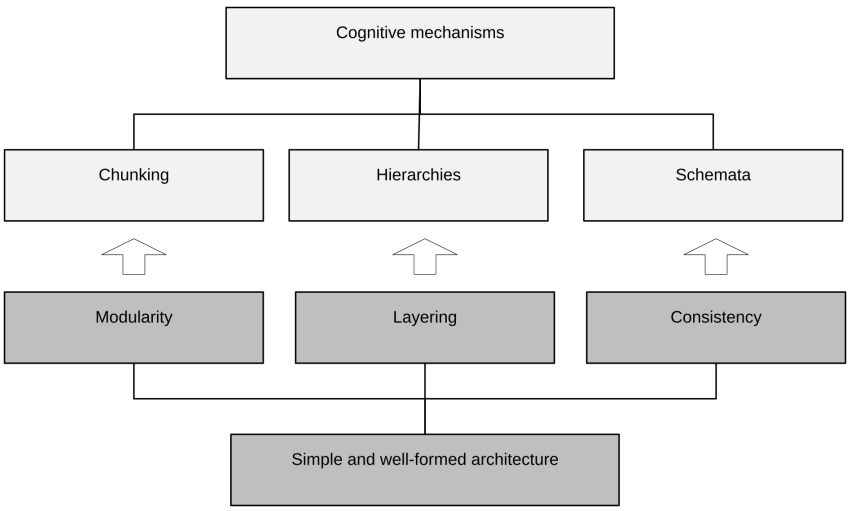
- You can't find all the information you need in the documentation or existing code - you must talk to people.

2.12 Cognitive Psychology and Architectural Principles

In the long period of evolution, the human brain has acquired some impressive mechanisms that help us to deal with complex structures. Cognitive psychology studies how the human mind processes information. It focuses on understanding the mental processes that underlie behavior, including perception, attention, language, problem-solving, and learning.

By understanding how the human mind processes information, we can create an architecture that are intuitive and easy to understand. Cognitive psychology can guide as with architectural principles, which are a set of guidelines or best practices that can be used to design of software systems.

These principles typically focus on the overall structure and organization of the system, and are intended to help ensure that the system is easy to maintainable, and easy to understand and modify. The goal should be to develop our software products for a long-term period with constant quality, also with varying development teams.



Cognitive Psychology and Architectural Principles

This mechanism our brain has developed for complex structures are **chunking**, **formation of hierarchies** and **building of schemata**. These techniques have direct mappings into the architectural design of a system. When these and their implementation in architecture are obvious to the development team, an essential base for high-quality architecture is set.

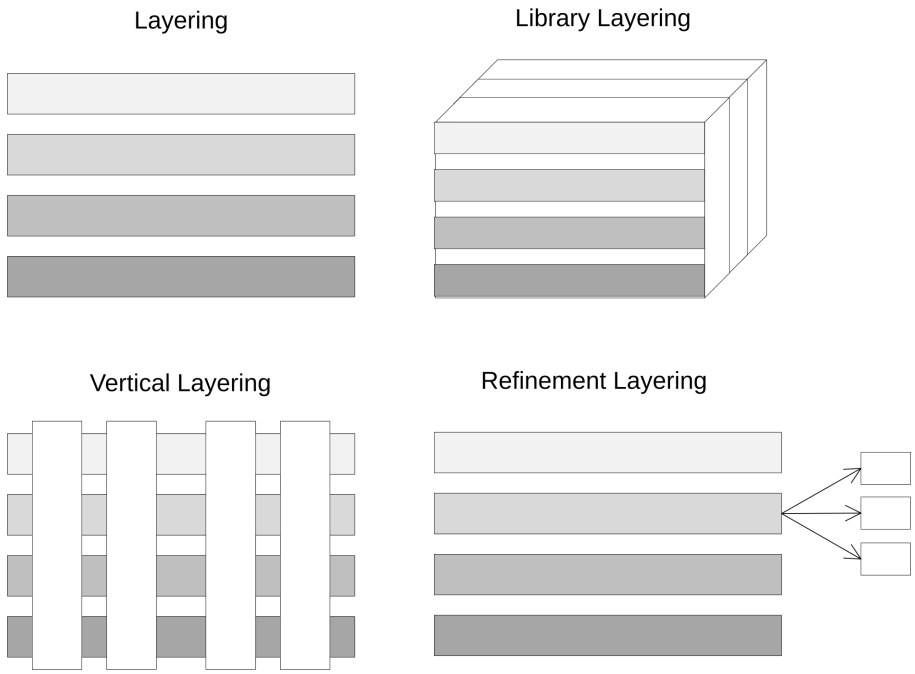
Carola Lilienthal described in her book *Sustainable Software Architecture, Analyze and Reduce Technical Debt* in detail how the cognitive mechanisms can be mapped to the software architecture. There you can find further details on the implementation of modularity, hierarchization, and pattern consistency.

2.13 Layered Architecture

2.13.1 Use of Layered Architecture

Architectural styles and patterns play an important role in software engineering. There are many possibilities for the architecture and design of an application. The complexity of an application depends on different requirements. A system should be designed in such a way that the architecture is easily reusable, extendable, and maintainable and allows a clear separation of components.

Architectures should not be about frameworks. Frameworks are tools to be used that support you in your desired design of an application. You should not be built your architecture around a framework.



Use of Layered Architectures

A common approach is the so-called n-tier architecture. Here, the building blocks of the application are divided into layers according to their task. This layering was the de-facto standard for many applications and therefore is widely known by most architects, designers, and developers.
Although the layer architecture pattern does not specify the number and types of layers, most layer architectures consist of three or four layers. For example, smaller applications may have only three layers, while larger and more complex business applications may have five or more layers.

A three-tier architecture consists of, for example, the persistence layer, the business layer, and the presentation layer. It includes clearly defined interfaces and a defined direction of the data flow. To minimize the coupling of the layers, domain objects are used to transport the data from layer to layer.

Traditional software architecture defines the following three main layers for a layered software design:

Presentation Layer

The task of the presentation layer is to respond to client requests. On the one hand, it presents the data on an interface in a proper form, and on the other hand, it includes the execution and reaction to user actions. When implementing and realizing this layer, the challenge is to keep it free of business logic. For this reason, the presentation layer should not implement business logic, but only call and use it.

Business Layer

The business layer represents a logical and presentation-independent layer. In it, the business logic is programmed, i.e., everything that must be done independently of a concrete presentation. It contains the entire functionality of the business requirements developed in the use cases. The focus during implementation is on the grouping of related components and the abstraction of the architecture to be developed.

Persistence Layer

The persistence layer is responsible for the actual access to the data in the database. The database access must not be performed by any other layer than the persistence layer to achieve the highest possible level of encapsulation. For the higher layers, it must be irrelevant in which kind the objects are stored. For this reason, no direct database call may be used in the business or presentation layer. The persistence layer is only called via the business objects of the business layer.

How do you organize your classes into packages?

Imagine you're writing a web application. In this application, you handle tickets and reservations. Your classes include classes like `TicketController`, `TicketService`, `TicketRepository` for the ticket handling and `RegistrationController`, `RegistrationService` and `RegistrationRepository` for registration. So, how do you organize your classes into packages and create a valid architecture?

There are two different ways to structure your packages. You can focus on the technical tiers:

- `com.swsc.web`
- `com.swsc.service`
- `com.swsc.repository`

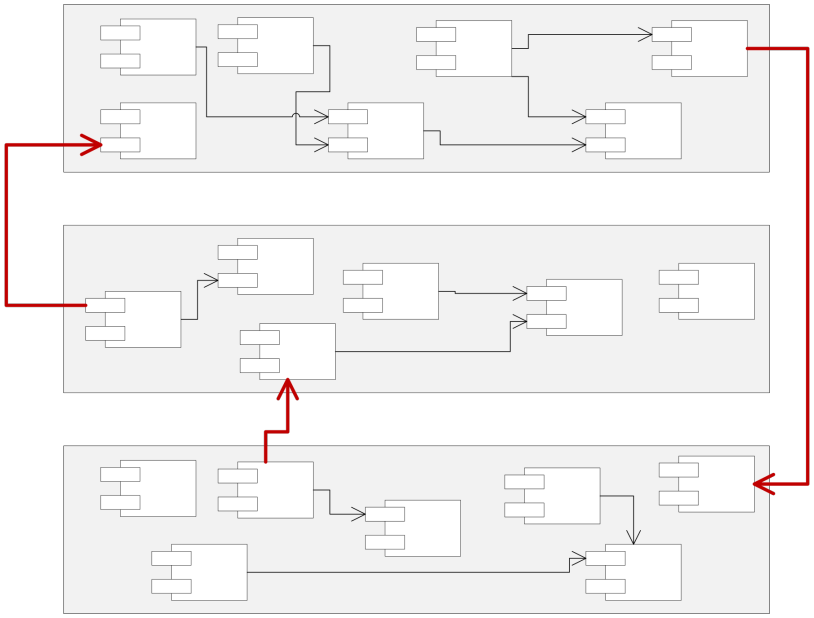
Or you can focus on the domain and separate the packages into feature-driven approach:

- `com.swsc.registration`
- `com.swsc.ticket`

This approach will be discussed in detail in the following chapters.

2.13.2 Violated Layered Architecture

In this example a request does not move from layer to layer, this violates the layered architecture. It must go through the layer right below it to get to the next layer below that one.



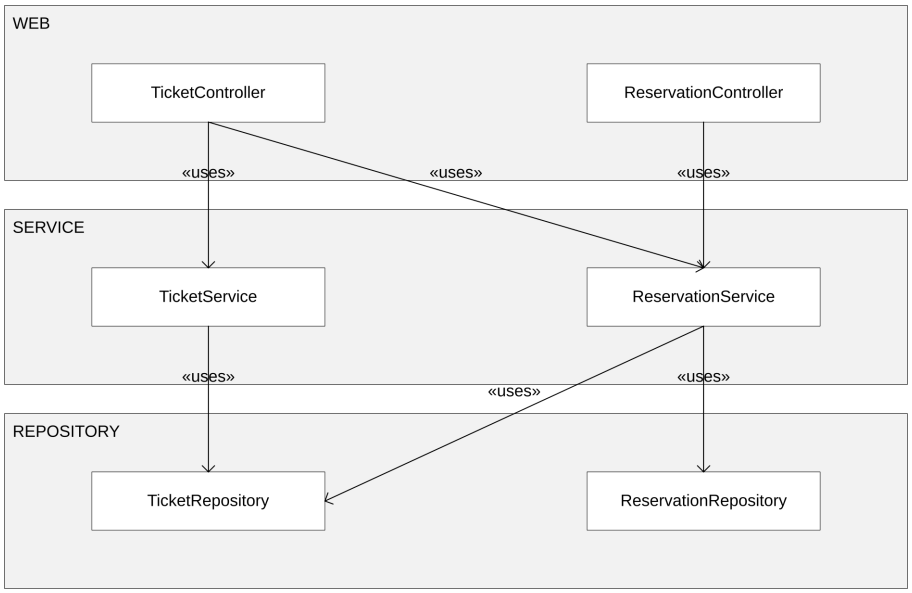
Violated Layered Architecture

The access of a lower layer to an upper one is also a violation in this example.

The access and the agreements concerning the architecture of an application should be known within the team or preferably checked automatically. A method that makes this possible is presented in the chapter *Testing the Architecture and Design*. This architecture should not disappear over time and should be valid.

2.13.3 Horizontal Layering

The classic horizontally layered architectural pattern was a solid general-purpose pattern and therefore a good choice for most applications, especially if you were not sure which architectural pattern was best for your application.



Horizontal Layering

Today, however, it is no longer the preferred solution. There are a few things to keep in mind when choosing this pattern from an architectural point of view. With this pattern, you run the risk of building a monolith. This pattern cuts the layers horizontally according to purely technical responsibility. This paradigm **eliminates low coupling and high cohesion**. This is exactly the opposite of what we want to achieve.

The different modules become quickly dependent on other parts of the application and are not separated from each other. If an application is structured purely according to horizontal layers and not according to features like in this case *Ticket* and *Reservation* you end up with a pure layer architecture without a clear division into feature-based modules.

When you look at the package structure you will recognize that almost all classes must have a `public` modifier to have access to the lower layer. Which is certainly not the best approach and violates the principle of information hiding.

Package structure - Horizontal Layering

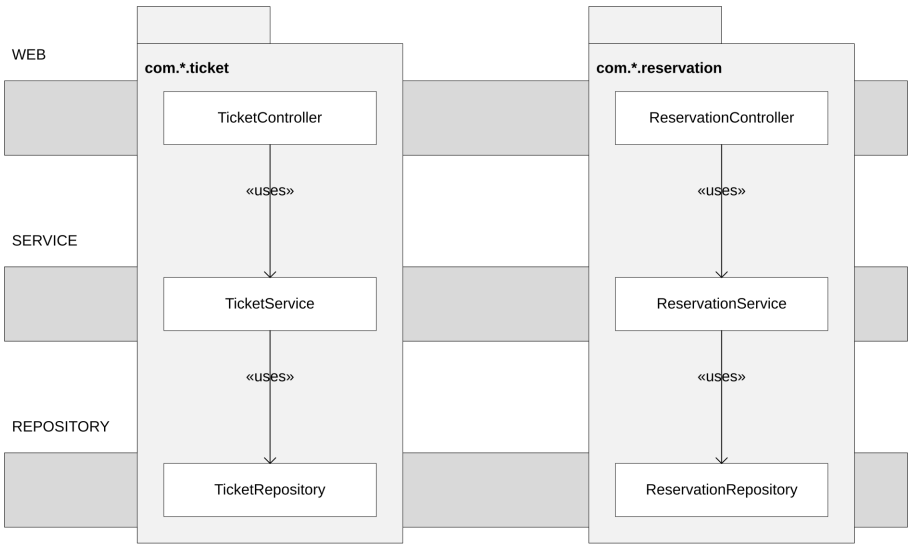
```
com.swcs.web
    (C) TicketController
    (C) ReservationController
com.swcs.service
    (C) TicketService
    (C) ReservationService
com.swcs.repository
    (C) TicketRepository
    (C) ReservationRepository
```

2.13.4 Feature-Based Layering - Single package

A better approach would therefore be to divide the application into **vertical slices** according to functional responsibilities and keep the technical layering as well. Each module is given responsibility for a clear and separate functional context. So, instead of coupling across a layer, we couple vertically along with a slice. Packaging by feature improves coupling and cohesion by keeping all classes related in the same package or module. This minimizes coupling between slices and maximizes coupling in a slice. In the long run, this decoupling in slices simplifies code maintenance because it is clear which part of the code does what. We can make changes without touching or fully understanding the rest of the code. It makes it easier to make changes to individual parts of the application, improving both maintainability and expandability.

The two concepts of cohesion and coupling are useful and important, but there is another point that comes into play in this approach.

The package and module structure by "Package by Feature" help the developer to see what it does and what functionality it has. The architecture jumps out at you. This is a step toward what Uncle Bob calls Screaming Architecture[3].



Feature Based Layering - Single package

With the single package approach, no sub-packages are used for the individual horizontal layers. This makes it possible to set all classes to package-private and thus prevent the use of these outside the package. However, this can affect the clarity of large modules.

Both horizontal and vertical slicing have their use cases, but it is obvious that vertical combined with horizontal slicing enables a more modern and adaptable architecture.

This approach makes it possible to reduce the visibility of the classes. They can be set to package-private and are therefore only available within the feature slices. In the horizontal layering, this was not possible because related classes had to communicate across packets, which forces us to make everything public. The horizontal layering within a slice is still present. The naming conventions like `*Controller`, `*Service` and `*Repository` define the layering.

[3] https://blog.cleancoder.com/uncle-bob/2011/09/30/Screaming-Architecture.html

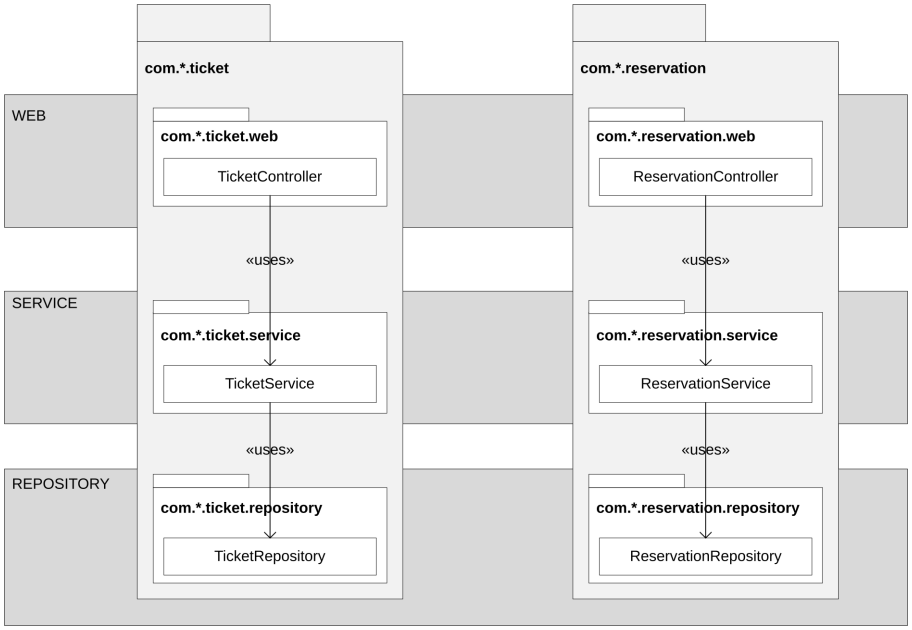
Package structure - Feature Based Layering - Single package

```
com.swcs.ticket
    (C) TicketController
    (C) TicketService
    (C) TicketRepository

com.swcs.reservation
    (C) ReservationController
    (C) ReservationService
    (C) ReservationRepository
```

The advantages are obvious:

- You get a low coupling and high cohesion of the slices.
- The code structure corresponds to the domain, which ensures a consistent understanding and communication between the developers and the department.
- As new features and enhancements are mostly technical driven, it is easier to identify the appropriate modules and to determine the level of change.
- Changes within a single functional domain can happen within a single part of the system. Side effects are avoided or reduced in this way because the modules are independent of each other.
- Vertical separation promotes forming hard contextual boundaries. Vertical layering enables easier architectural transformation from a monolithic structure to a distributed structure built like micro-services.
- Testability is better because the technical logic of a module is separate from other modules.

2.13.5 Feature-Based Layering - Slices before layers



Feature Based Layering - Slices before layers

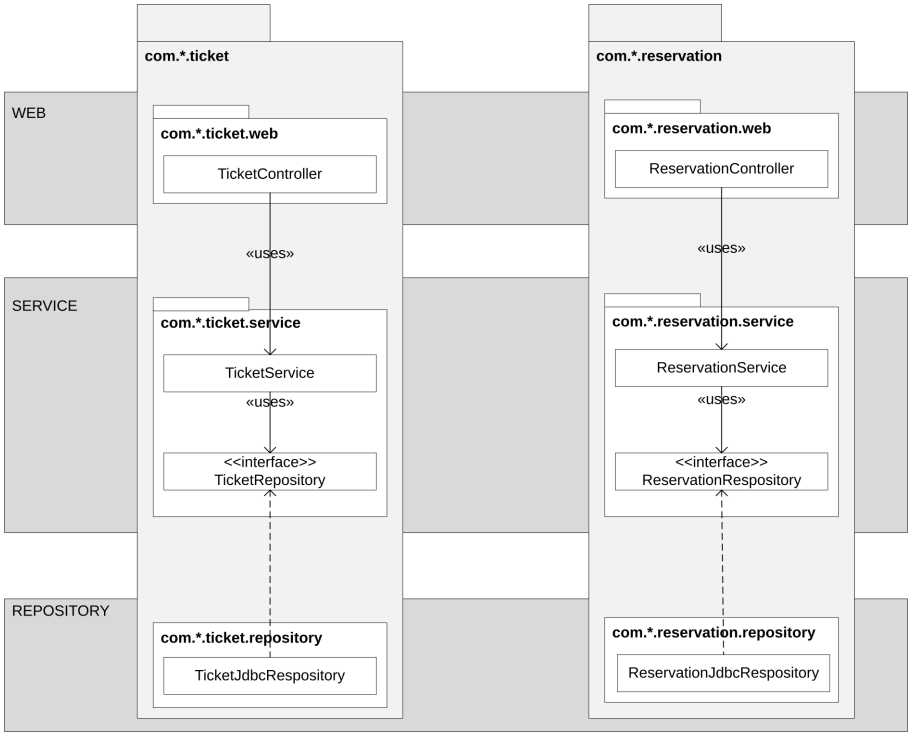
The approach of *Slices before layers* divides the vertically feature slice into technical layers such as web, service, and repository. This makes the technical layers within the package easier to recognize. This serves mainly the organization of the codebase and improves the structure of the code.

However, a very important advantage is lost. To access each other, the classes have to be marked with public with the consequence that they are visible for other feature slices.

Package structure - Feature Based Layering - Slices before layers

```
com.swcs.ticket.web
    (C) TicketController
com.swcs.ticket.service
    (C) TicketService
com.swcs.ticket.repository
    (C) TicketRepository

com.swcs.reservation.web
    (C) ReservationController
com.swcs.reservation.service
    (C) ReservationService
com.swcs.reservation.repository
    (C) ReservationRepository
```

2.13.6 Feature-Based Layering – Hexagonal Architecture



Feature Based Layering - Hexagonal Architecture

The hexagonal architecture[4], or port and adapter architecture, is an architectural pattern used in software design to structure large applications. It is designed to create loosely coupled application components by using ports and adapters.

This architectural style can also be applied to code within a module. The principle behind it is that the domain logic is centrally encapsulated. The interaction with the user and server-side with this domain lies outside the core. The user side can be an API or a web interface. The server-side can be a database. The key point here is that the outside world always depends on the domain core and never the other way around.

In the example, the service package defines the interfaces like `TicketRepository` for the actual access logic to the database. The actual implementation of the interface `TicketJdbcRepository` is located in the package repository and therefore points in the direction of the domain package.

Package structure - Feature Based Layering - Hexagonal Architecture

```
com.swcs.ticket.web
    (C) TicketController
com.swcs.ticket.service
    (C) TicketService
    (I) TicketRepository
com.swcs.ticket.repository
    (C) TicketJdbcRepository

com.swcs.reservation.web
    (C) ReservationController
```

[4] https://alistair.cockburn.us/hexagonal-architecture/

```
com.swcs.reservation.service
    (C) ReservationService
    (I) ReservationRepository
com.swcs.reservation.repository
    (C) ReservationJdbcRepository
```

The advantage of this approach is that the entire business logic is defined centrally in a business component and is decoupled from the outside world. But the visibility here is also too much open.

Reducing dependencies with application events

Direct dependencies between modules are usually created by direct calls and the use of classes from another module. Dependencies should be reduced as far as possible. Cyclic dependencies should be avoided at all costs.

There are several ways to reduce them like application events. Application events can be used for loosely coupled components to exchange information. By replacing the direct call with an application event, you can reverse the dependencies. A module creates an event of a certain type, which is located within the module. Another module registers an event listener, which listens and reacts to this event type. The received module thus has a dependency on the sending module, because it knows this event type and not vice versa. This is especially useful for calls with direct commands to another module like audit logs.

2.13.7 The Java Module System

With the Java Module System[5] introduced in Java 9, you get another way to add visibility rules to your architecture. The above examples were designed without the capabilities of the module system. But if you use this, it will make it easier for you to organize your code.

The modularity adds a higher level of aggregation above the packages. The packages in one module are only accessible to other modules if the module explicitly exports them. Even then, another module can only use these packages if it explicitly declares that it needs the functionality of the other module.

You will find that taking the module system into account helps you to develop cleaner, more consistent designs. You don't need a modular system to design for modularity, but a modular system makes this much easier.

The book *The Java Module System* by Nicolai Parlog gives a detailed insight into the features of the module system.

[5] https://openjdk.java.net/jeps/261

2.14 Architecture Documentation

Designing architecture means making decisions! The documentation is a persistent and communicable artifact for all decisions. It is taken into account during the planning of the project and focuses more on non-functional than functional requirements. The team works closely together to build a system that meets all requirements. Some development work will address needs other than functional requirements. Therefore, the work on improving the quality of the system must take up a substantial part of the team's time.

To give all stakeholders of a system, that require information about the architecture, e.g., the internal structure, crosscutting concepts, or fundamental decisions a fundamental architecture overview, it is important to provide architecture documentation.

Arc42[6] provides a comprehensive but condensed template for describing architectures. This template produces uniformed architecture documentation which helps to understand the big picture and connect the dots by looking at multiple areas.

The template is structured as follows and contains 12 structure items[7]:

1. Introduction and Goals

Short description of the requirements, driving forces, extract of requirements. Top three or max five quality goals for the architecture which have the highest priority for the major stakeholders. A table of important stakeholders with their expectations regarding architecture.

2. Constraints

Anything that constrains teams in the design and implementation decisions or decisions about related processes. Can sometimes go beyond individual systems and are valid for whole organizations and companies.

3. Context and Scope

Delimits your system from its external communication partners. Specifies the external interfaces.

4. Solution Strategy

Summary of the fundamental decisions and solution strategies that shape the architecture. Can include technology, top-level decomposition, approaches to achieve top-quality goals, and relevant organizational decisions.

5. Building Block View

Static decomposition of the system, abstractions of source code, up to the appropriate level of detail.

6. Runtime View

The behavior of building blocks as scenarios, covering important use cases or features, interactions at critical external interfaces, operation and administration plus error and exception behavior.

7. Deployment View

[6] https://arc42.org/
[7] https://arc42.org/overview/

Technical infrastructure with environments, computers, processors, topologies. Mapping of software building blocks to infrastructure elements.

8. Crosscutting Concepts

Overall, principal regulations and solution approaches are relevant in multiple parts of the system.

9. Architectural Decisions

Important, expensive, critical, large-scale, or risky architecture decisions including rationales.

10. Quality Requirements

Quality requirements as scenarios, with a quality tree to provide a high-level overview.

11. Risks and Technical Debt

Known technical risks or technical debt.

12. Glossary

Important domain and technical terms that stakeholders use when discussing the system.

2.15 Testing the Architecture and Design

The architecture of a system usually is a design that was made at a certain moment in time. However, as the system evolves and developers come and go, it's hard to keep everyone up to date on a reference architecture or encourage developers to make changes to it, since it's usually in the form of documents, confluence pages, or something similar. This can result in code that violates architectural specifications increasing the technical debt.

IntelliJ[8]
: IntelliJ comes with a simple tool for architecture analysis. With it, you can generate a dependency matrix for your code.

Eclipse[9]
: In eclipse you can install various plugins to visualise dependencies and to detect cycles. eDepend[10], STAN[11], jDepend[12], Java Dependency Viewer[13]

ArchUnit[14]
: With ArchUnit you can check the architecture constraints of your Java application automatically and can run as part of a test suite. It is a free, simple, and extensible library for checking the architecture of Java code, which can check dependencies between packages and classes, layers and slices, cyclic dependencies, and more.

JQAssistant[15]
: jQAssistant is a QA tool, which allows the definition and validation of project-specific rules on a structural level. It is built upon the graph database Neo4j and can easily be plugged into the build process to automate the detection of constraint violations and generate reports about user-defined concepts and metrics.

Structure 101[16]
: With Structure 101 you can understand, analyze, (drag 'n drop) refactor, and control a large, complex codebase.

Sonargraph[17]
: Sonargraph is a powerful static code analyzer that allows you to monitor a software system for technical quality and enforce rules regarding software architecture, metrics, and other aspects in all stages of the development process.

Lattix[18]
: Lattix enables you to quickly identify and remediate architectural issues. It gives software architects a fast and visual way to represent an application's architecture with the Dependency Structure Matrix technology.

Teamscale[19]
: Teamscale supports your team to analyze, monitor, and improve the quality of your software.

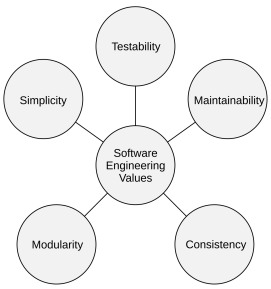
[8] https://www.jetbrains.com/
[9] https://www.eclipse.org/
[10] https://marketplace.eclipse.org/category/free-tagging/edepend-graphical-dependency-analysis-tool-340
[11] http://stan4j.com/eclipse/
[12] https://marketplace.eclipse.org/category/free-tagging/jdepend
[13] https://marketplace.eclipse.org/content/java-dependency-viewer
[14] https://www.archunit.org/
[15] https://jqassistant.org/
[16] https://structure101.com/
[17] https://www.hello2morrow.com/products/sonargraph
[18] https://www.lattix.com/
[19] https://www.cqse.eu/en/teamscale/overview/

CodeScene[20]

Is a multi-purpose tool bridging code, business, and people. See hidden risks and social patterns in your code. Prioritize and reduce technical debt.

[20] https://codescene.com/

2.16 Software Engineering Values



Software Engineering Values

Testability

- To ensure the functional correctness of the system, it is essential that each unit of the system is verifiable and has been tested.
- A system is only complete if its functional correctness and completeness are proven by automatically triggered tests.

Maintainability

- We build durable systems, which means we always must consider how to maintain the systems.
- The colleagues who maintain a system can change over time. Therefore, a maintainable system should be easy to understand. The level of system-specific knowledge required to carry out maintenance should be kept to a minimum.

Consistency

- To improve flexibility and maintenance, the number of different solutions like code-level patterns, libraries, third-party services, and application integration patterns, for the same problem should be kept to a minimum.
- Having multiple solutions to a problem is better than having no solutions, having a single solution to a problem is the best.

Modularity

- Things are always changing. To protect our code from this fundamental fact, we need to limit the impact of change and divide it into reasonably independent, coherent units at all levels.

Simplicity

- The simplest parts of a system to change are those that do not exist. For any given requirement, the simplest, minimalist solution should be provided which meets the previous values.

2.17 Team Charter

A team charter is a document that clarifies the direction of the team by setting a **vision**, **mission**, and **boundaries** that everyone on the team supports. As a source, it illustrates the focus and direction of the team to others in the organization. The charter should be developed within the team to promote clear understanding and agreement, providing clarity and reducing confusion in cases where conflicting tasks or projects arise.

A Team Charter could look like the following example:

From a team-facing practice view, we want

- create an open, no blame, learning, and feedback culture
- create an environment that fosters teamwork and innovation
- create collective ownership with no knowledge silos
- developing and utilizing everyone's abilities to the fullest
- communicate effectively and talk to each other
- provide honest feedback, act upon the feedback, and drive for continuous improvement
- collaborate and do pair programming at a level that drives the highest quality
- be a whole team and the unit of delivery
- be a professional software craftsman

From a technical practice view, we want

- keep things simple
- encourage continual improvement through refactoring
- closing the feedback loop frequently through CI/CD
- that everybody helps to fix the build immediately
- monitor CI/CD/SonarQube after push
- keep our products free of technical debt
- follow our standards and best practices
- enforce the DoD and our Code review Checklist

3. Clean Code Best Practices

3.1 Communicate through code

There are only two hard things in Computer Science: cache invalidation and naming things.
– Phil Karlton

3.1.1 Use Java code conventions and avoid misinformation

Java code conventions are a set of guidelines for writing Java code that are intended to help improve the readability and maintainability of the code. Some of the key conventions include using descriptive and meaningful names for variables and methods, using consistent indentation and spacing, and using comments to document the code.

To avoid misinformation, it is important to follow this code conventions. This can help to ensure that other developers who may be working on the code in the future can easily understand how the code works and what it does.

Bad code

```
1  public static final double pi = 3.14159265358979323846;
2  int YEAR;
3  String first_name;
4  public class convert_strategy {}
5  Set carList;
```

In this code snippet, there are several violations of Java code conventions and potential sources of misinformation:

- **Inconsistent Naming**: The variable `pi` violates the convention of using uppercase letters and underscores for constant names. It should be renamed to `PI` to adhere to the convention. Similarly, the variable `YEAR` violates the convention of using lowercase letters for variable names. It should be renamed to year for consistency.
- **Camel Case Naming**: The variable `first_name` violates the convention of using camel case for variable names. It should be renamed to `firstName` to follow the convention.
- **Class Naming**: The class `convert_strategy` violates the convention of using PascalCase for class names. It should be renamed to `ConvertStrategy` to adhere to the convention.
- **Inappropriate Naming**: The variable `carList` is misleading because it implies that it is a list of cars. However, it is declared as a `Set`, which may cause confusion for other developers. It should be renamed to `carSet` or another appropriate name that reflects the actual data structure being used.

Good code

```
public static final double PI = 3.14159265358979323846;
int year;
String firstName
public class ConvertStrategy {}
Set cars;
```

Bad code

```
Customer Customer = new Customer("Darth Vader");
...
Customer.name() // static access?
```

In this code, there are a couple of issues related to naming conventions and potential misinformation:

- **Inconsistent Naming**: The variable `Customer` violates the convention of using lowercase letters for variable names. It should be renamed to `customer` to follow the convention and differentiate it from the class name.
- **Static Access Misconception**: The line `Customer.name()` suggests that `name()` is a static method, which could lead to misinformation. However, if `Customer` is an instance of the `Customer` class, it should be called as `customer.name()` to access the instance method. Using proper naming conventions and avoiding misleading code can prevent such misunderstandings.

Good code

```
Customer customer = new Customer("Darth Vader");
...
customer.name()
```

Java code conventions can be found here:
Oracle Java code conventions[1]
Google Java Style Guide[2]

[1] https://www.oracle.com/technetwork/java/codeconventions-150003.pdf
[2] https://google.github.io/styleguide/javaguide.html

3.1.2 Choose an expressive name and avoid mental mapping

Choose an expressive name for a variable, method, class, or other code element that accurately and clearly describes its purpose or behavior. This can help to make the code more readable and understandable, and can reduce the need for other developers to mentally map the code elements to their intended meanings.

Bad code

```
int d; // days since birthday
String n; // name of host
```

In this code, the variables `d` and `n` are not very descriptive. It is not immediately clear what these variables represent or what they are used for. As a result, other developers who may be working on this code in the future may need to spend time mentally mapping these variables to their intended meanings, which can be time-consuming and error-prone.

Good code

```
int daysSinceBirthday;
String hostname;
```

In this code, the variables `daysSinceBirthday` and `hostname` are more expressive. Their names accurately describe their intended meanings and purposes, which makes it easier to understand the code without having to mentally map the variables to their meanings.

3.1.3 Make differences clear with meaningful variable names

Bad code

```
public boolean isFileOlder(File s1, File s2) {}

public static String replaceFirst(String string, Pattern pattern, String with) {}
```

Good code

```
public boolean isFileOlder(File file, File referenceFile) {}

public static String replaceFirst(String text, Pattern regex, String replacement) {}
```

3.1.4 Use pronounceable names

Use pronounceable names for variables, methods, classes, and other code elements that are easy to pronounce and remember. This can make the code more readable and understandable, and can make it easier for other developers to work with the code.

Bad code

```
class DRcd {
     private int rtd;
     private String stna;
     private LocalDateTime genymdhms;
     private LocalDateTime modymdhms;
}
```

In this code, the variables `rtd`, `stna`, `genymdhms` and `modymdhms` are not very pronounceable. Their names are not easy to say out loud, and they do not convey any meaning or information about their intended purposes. As a result, other developers who are working with this code may find it difficult to remember or refer to these variables.

Good code

```
class Student {
     private int registrationId;
     private String name;
     private LocalDateTime generated;
     private LocalDateTime modified;
}
```

3.1.5 Do not hurt the readers

We will read more code than we write, so do not hurt the readers. Declare class variables with useful constants. In this way, the meaning or intended use of each literal is indicated. If the constant needs to be changed, the change is also limited to the declaration. There is no need to search and change the code for this literal.

Bad code

```
// What is 86400000?
setTokenTimeout(86400000);
```

Good code

```
private static final int MILLISECONDS_IN_A_DAY = 24 * 60 * 60 * 1000;

setTokenTimeout(MILLISECONDS_IN_A_DAY);
```

Bad code

```
public final class Sphere {

    private Sphere() {
    }

    public static double area(double radius) {
        return 3.14 * radius * radius;
    }

    public static double volume(double radius) {
        return 4.19 * radius * radius * radius; // Why 4.19?
    }
}
```

Good code

```
public final class Sphere {

    private Sphere() {
    }

    public static double area(double radius) {
        return Math.PI * radius * radius;
    }

    public static double volume(double radius) {
        return 4.0 / 3.0 * Math.PI * radius * radius * radius;
    }
}
```

3.1.6 Don't add redundant context

Do not add redundant context in variable or class names that include unnecessary information or details that can make the names longer and less readable.

Bad code

```
class CarData {
     private String carManufacturer;
     private String carModel;
}
```

In this code, the `carManufacturer` and the `carModel` variable both include the word `car` as part of their names. This is redundant, because the names already include the word "CarData" as part of the class name, so it is clear that they are related to cars. As a result, the names are longer and less readable than they need to be.

Moreover, the name of the class `CarData` could be changed to `Car` as `Data` adds no additional meaning to the class name.

Good code

```
class Car {
     private String manufacturer;
     private String model;
}
```

Bad code

```
public interface UserRepository {

    Optional<User> findUserById(long userId);

    Iterable<User> findAllUsers();

    User saveUser(User user);

    Iterable<User> saveAllUsers(Iterable<User> users);

    void deleteUserById(long userId);

    void deleteAllUsers();

    long countUsers();
}
```

Good code

```
public interface UserRepository {

    Optional<User> findById(long id);

    Iterable<User> findAll();

    User save(User user);

    Iterable<User> saveAll(Iterable<User> users);

    void deleteById(long id);

    void deleteAll();

    long count();
}
```

3.1.7 Don’t add words without additional meaning

Some generic words often don’t add any additional meaning to a name. So, they mean anything when added to a name.

These words often indicate this constellation:

- Data
- Information
- Bean
- Value
- Manager
- Object
- Entity
- Instance
- Custom

By adding these kinds of words, double-check if the name still means the same if you remove it? If yes, remove it. If not, keep it, or better try to find a better name.

Bad code

```
public final class StudentData {
    private final String studentFirstName;
    private final String studentLastName;
    private final AddressInformation addressInformation;
    private final CoursesBean coursesBean;

    public StudentData(String studentFirstName, String studentLastName,
            AddressInformation addressInformation, CoursesBean coursesBean) {
        this.studentFirstName = studentFirstName;
        this.studentLastName = studentLastName;
        this.addressInformation = addressInformation;
        this.coursesBean= coursesBean;
    }
}
```

Good code

```
public final class Student {
    private final String firstName;
    private final String lastName;
    private final Address address;
    private final Courses courses;

    public Student(String firstName, String lastName, Address address, Courses courses) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.address = address;
        this.courses = courses;
    }
}
```

3.1.8 Don't use *and* or *or* in method names

A method should be responsible only for one action. Method names using *and* or *or* are an indicator, that the method has too many responsibilities and combines too much functionality. In this case, you should consider splitting the functionality of this method into other methods.

How to improve this:

- Split the method into smaller pieces and extract the method in separate methods.
- If the things belong together:
  - Consider finding a better name for that entire operation.
  - Consider creating something else that encapsulates the combination.

Bad code

```
public class JavaMailSender {
    public void createMimeMessageAndSend() throws MailException {}
    ...
}
```

Good code

```
public class JavaMailSender {
    public MimeMessage createMimeMessage() {}
    public void send(MimeMessage mimeMessage) throws MailException {}
    ...
}
```

3.1.9 Use positive names for boolean variables and functions

When defining boolean functions, it is often best to use positive names that accurately describe their intended behavior or state. For example, instead of defining a function named `isNotOpen()`, it would be more intuitive and readable to define a function named `isOpen()`. This makes the code more self-explanatory and easier to understand.

Bad code

```
if (!file.isNotOpen())
if (!window.isNotVisible())

if (!notRunning)
if (!hasNoLicense)
```

Here, the functions `isNotOpen()`, `isNotVisible()` and the boolean variables `notRunning` and `hasNoLicense` are defined with negative names, which can lead to confusion and make the code harder to comprehend. Negated names require extra mental processing to understand their actual meaning. It is preferable to use positive names that convey the intended behavior more directly.

Good code

```
if (file.isOpen())
If (window.isVisible())

if (running)
if (hasLicense)
```

By using positive names for boolean variables and functions, you create code that is more intuitive, easier to understand, and less prone to misinterpretation. Positive names improve code readability and reduce cognitive load for developers who read or maintain the code, leading to more efficient and error-free programming.

3.1.10 Respect the order within classes

- Be consistent with the order of the class attributes, the constructor parameters, and its methods.
- Align the order of the constructor parameter and the methods with the order of the class attributes declaration order.
- Private methods should be below the calling method. Thus, the code tells a story with the related methods below it. Furthermore, searching and scrolling are prevented.

Bad code

```
public final class WebServer {

    private final int port;
    private final String hostname;

    public WebServer() {
        this("localhost", 8080);
    }

    public WebServer(String hostname, int port) {
        this.port = port;
        this.hostname = hostname;
    }

    public String hostname() {
        return this.hostname;
    }

    public void start() {
        if (isPortFree()) {
           ....
        }
    }

    public int port() {
        return this.port;
    }

    public void stop() {
    ...
        destroyThreadPool();
        ...
    }

    private boolean isPortFree() { ... }

    private void destroyThreadPool() { ... }
}
```

Good code

```
public final class WebServer {

    private final int port;
    private final String hostname;

    public WebServer() {
        this(8080, "localhost");
    }

    public WebServer(int port, String hostname) {
        this.port = port;
        this.hostname = hostname;
    }

    public int port() {
        return this.port;
    }

    public String hostname() {
        return this.hostname;
    }

    public void start() {
        if (isPortFree()) {
           ....
        }
    }

    private boolean isPortFree() { ... }

    public void stop() {
    ...
        destroyThreadPool();
        ...
    }

    private void destroyThreadPool() { ... }
}
```

3.1.11 Group by line break

Use line breaks to make associations with related things and separate weakly related things.

Bad code

```
public final class FileUtils {
    public static final long ONE_KB = 1024;
    public static final long ONE_MB = ONE_KB * ONE_KB;
    public static final long ONE_GB = ONE_KB * ONE_MB;
    public static final File[] EMPTY_FILE_ARRAY = new File[0];
    private FileUtils(){}
    public static void cleanDirectory(File directory) throws IOException {
        File[] files = verifyFiles(directory);
        List<Exception> causes = new ArrayList<>();
        for (File file : files) {
            try {
                forceDelete(file);
            } catch (final IOException ioe) {
                causes.add(ioe);
            }
        }
        if (!causes.isEmpty()) {
            throw new IOException(causes);
        }
    }
}
```

Good code

```
public final class FileUtils {

    public static final long ONE_KB = 1024;
    public static final long ONE_MB = ONE_KB * ONE_KB;
    public static final long ONE_GB = ONE_KB * ONE_MB;

    public static final File[] EMPTY_FILE_ARRAY = new File[0];

    private FileUtils() {}

    public static void cleanDirectory(File directory) throws IOException {
        File[] files = verifyFiles(directory);
        List<Exception> causes = new ArrayList<>();

        for (File file : files) {
            try {
                forceDelete(file);
            } catch (final IOException ioe) {
                causes.add(ioe);
            }
        }

        if (!causes.isEmpty()) {
            throw new IOException(causes);
        }
    }
}
```

3.1.12 Prefer self-explanatory code instead of comments

Don't comment bad code - rewrite it.
–Brian W. Kernighan, The Elements of Programming Style

Prefer self-explanatory code that is clear and easy to understand without the need for additional comments or explanations.

- Comments are no substitute for bad code.
- Explain it through code.
- Java is a programming language and should express itself.
- The more self-explanatory the code is, the less you need to comment it.
- Treat a comment as a danger signal.
- Comments are like deo for stinky code.
- Comment **WHY** not **WHAT**!

Bad code

```
final class InchToPointConvertor {

    private InchToPointConvertor{}

    // convert the quantity in inches to points
    static float parseInch(float inch) {
       return inch * 72; // one inch contains 72 points
    }
}
```

Good code

```
final class InchToPointConvertor {

    private static final int POINTS_PER_INCH = 72;

    private InchToPointConvertor{}

    static float toPoints(float inch) {
        return inch * POINTS_PER_INCH;
    }
}
```

Bad code

```
// Check if employee has entitlement for extra vacation days
if ((employee.noVacationDaysLeft && employee.age > 50)
   || (employee.noVacationDaysLeft && employee.hasChildren))
```

Good code

```
if (employee.hasEntitlementForBonusVacation())
```

Bad code

```
class Passwords {
    ...
    // check if the password is complex enough
    public boolean isComplexPassword(String password) {
        boolean symbolOrDigitFound = false; // found a digit or symbol?
        boolean letterFound = false; // found a letter?

        for (int i = 0; i < password.length(); i++) {
            char c = password.charAt(i);

            if (Character.isLowerCase(c) || Character.isUpperCase(c)) {
                letterFound = true;
            } else {
               symbolOrDigitFound = true;
            }
        }

        return letterFound && symbolOrDigitFound;
    }
}
```

Good code

```
class Passwords {
    ...
    public boolean isComplex(String password){
        return containsLetter(password) && (containsDigit(password) || containsSymbol(password));
    }

    private boolean containsLetter(String password) { ... }

    private boolean containsDigit(String password) { ... }

    private boolean containsSymbol(String password) { ... }
}
```

3.1.13 Refactor step by step

Bad code

```
public List<int[]> getList() {
    List<int[]> data = new ArrayList<int[]>();
    for (int[] x : items)
        if (x[0] == 4) // 4 represents flagged
            data.add(x);
    return data;
}
```

Better code

```
public List<int[]> flaggedCells() {
    List<int[]> flaggedCells = new ArrayList<int[]>();
    for (int[] cell : gameBoard) {
        if (cell[STATUS] == FLAGGED)
            flaggedCells.add(cell);
    return flaggedCells;
}
```

Good code

```
public List<Cell> flaggedCells() {
    List<Cell> flaggedCells = new ArrayList<Cell>();
    for (Cell cell : gameBoard) {
        if (cell.isFlagged()) {
            flaggedCells.add(cell);
        }
    }
    return flaggedCells;
}
```

Craftsman code

```
public List<Cell> flaggedCells() {
    return gameBoard.stream()
            .filter(Cell::isFlagged)
            .toList();
 }
```

More generic craftsman code

```
public List<Cell> cells(Predicate<Cell> filter) {
    return gameBoard.stream()
            .filter(filter)
            .toList();
}
```

3.2 Bad comments

3.2.1 Redundant comments

Bad code

```
// Deletes a directory and checks if it is a directory.
// Throw IOException if not exists.
public static void deleteDirectory(File directory) throws IOException {
   if (directory != null && directory.isDirectory() && !directory.exists()) {
        return;
   }

   if (!directory.delete()) {
        throw new IOException("Unable to delete directory " + directory + ".");
   }
}
```

- Reading the comment probably takes longer than reading the code itself.
- The comment is certainly no more informative than the code.
- Neither does it justify the code, nor does it state its purpose or reason for existence.
- The commentary is less precise than the code and seduces the reader to accept this lack of precision.

3.2.2 Misleading comments

Bad code

```
// Deletes a directory and checks if it is a directory.
// Throw IOException if not exists.
public static void deleteDirectory(File directory) throws IOException {
   if (directory != null && directory.isDirectory() && !directory.exists()) {
        return;
   }

   if (!directory.delete()) {
        throw new IOException("Unable to delete directory " + directory + ".");
   }
}
```

- This comment is redundant and misleading!
- This method does not throw `IOException` if the directory does not exist. It throws `IOException` if the directory cannot be deleted.

3.2.3 Mandatory comments

Bad code

```
/**
 * Returns the day of the year.
 *
 * @return the day of the year.
 */
public int getDayOfYear() {
    return this.dayOfYear;
}
```

Bad code

```
/**
* @param title The title of the CD
* @param author The author of the CD
* @param tracks The number of tracks on the CD
* @param durationInMinutes The duration of the CD in minutes
*/
public void addCD(String title, String author, int tracks, int durationInMinutes)
```

- This commentary provides no additional information, only obscures the code and brings the potential for misleading.
- It's bad to rule that every function of a Javadoc or every variable should have a comment.
- Such comments only make the code more confusing and lead to general confusion and disorder.

3.2.4 Diary comments

Bad code

```
// Changes history
//
// 2020-09-30 Implement FactoryAware to inject Factory
// 2020-09-30 Remove unnecessary stubbing
// 2020-09-30 Adapt to API changes
// 2020-09-30 Merge branch '2.3.x'
// 2020-09-30 Introduce a dedicated @Compatibility annotation
// 2020-09-30 Fix matching of SNAPSHOT artifacts when customizing layers
// 2020-09-30 Adapt to API change
// 2020-09-30 Merge branch '2.3.x'
// 2020-09-30 Start building against version 5.3.0
// 2020-09-30 Do not execute datasource initialization in a separate thread
// 2020-09-30 Start building against Java 16
// 2020-09-29 Reduce configuration resolution when building a layered jar
// 2020-09-29 Merge pull request #56701
// 2020-09-24 Polish
// 2020-09-17 Add support for Oracle
```

- A long time ago, there was a reason for the creation and maintenance of such log entries.
- Nowadays there are version control systems.
- These comments are disturbance data that spread confusion and should be removed.

3.2.5 Gossip

Bad code

```
/**
 * Default constructor.
 */
public Parser () {
}
```

Bad code

```
/** The day of the month. */
private int dayOfMonth;
```

Bad code

```
private void startSending() {
    try {
        send();
    } catch (Exception e) {
        try {
            response.add(ErrorResponder.makeExceptionString(e));
            response.closeAll();
        } catch (Exception e1) {
            // Who cares!?
        }
    }
}
```

- These comments are so garrulous that we learn to ignore them.
- When we read the code, our eyes just jump over it.
- At some point, the comments start lying when the surrounding code changes.

3.2.6 Position identifier

Bad code

```
// Methods ////////////////////////////////////
```

Bad code, seen in a real project

```
//*************************************************
//***   getter and setter
//*************************************************
```

3.2.7 Write-ups and incidental remarks

Bad code

```
// Added by Bad Programmer
```

- This information can be seen in the version management system.

3.2.8 Don’t leave commented out code in your codebase

Bad code

```
public static String extension(String filename) {
   // if (filename == null) { return null; }

   String name = new File(filename).getName();
   int extensionPosition = name.lastIndexOf('.');
   if (extensionPosition < 0) {
      return "";
   }
   return name.substring(extensionPosition + 1);
}
```

- Others who see this commented code will not have the courage to delete it.
- They will believe that there is a reason why the code is there and that it is too important to be deleted.

3.2.9 Rules for commenting

Primary Rule

Comments are for things that can **not** be expressed in code.

Redundancy Rule

Comments which **restate** code must be deleted.

Single Truth Rule

If the comment says what the code **could** say, then the code must change to make the comment redundant.

More information can be found here:
https://agileinaflash.blogspot.com/2009/04/rules-for-commenting.html

3.3 Classes and objects

Designing Java classes is crucial for creating effective and reusable software. Follow these best practices when designing Java classes:

- **Standard Java Code Conventions**: Maintain readability and consistency by adhering to the standard Java code conventions. Use proper indentation, naming conventions, and comment code as necessary.
- **Single Responsibility**: Each Java class should have a clear and single purpose. Avoid combining too many responsibilities within a single class, as it can lead to confusion and maintenance difficulties.
- **Flexibility and Extensibility**: Design classes to be flexible and easily adaptable to future changes. Ensure that new features and functionality can be seamlessly integrated into the existing class without significant modifications.
- **Consistency**: Maintain consistency in your class design throughout your project. Use consistent naming conventions, coding standards, and design patterns. Consistency improves code readability and makes it easier for others to understand and work with your code.
- **Principle of Least Astonishment (POLA)**: Design classes with intuitive and predictable behavior. Users should not be surprised by the behavior of the system. Follow standard conventions and design principles to minimize unexpected behavior.

In conclusion, to design effective Java classes, keep them small, focused on a single responsibility, and use access modifiers wisely.

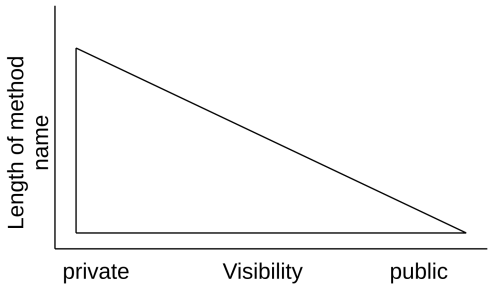
According to *Clean Code* by Robert C. Martin, the following best practices can help you write good classes, functions, and variables that are easy to read and modify.

3.3.1 Classes

- Use noun or noun phrase names for classes.
- Avoid naming classes as verbs.
- Hide internals and minimize the scope of variables and methods.
- Follow the Law of Demeter (LoD) - a class should only have knowledge of its direct dependencies.
- Base classes should not have knowledge of their derivatives.
- Prefer immutability whenever possible.
- Use dependency injection to decouple dependencies.

3.3.2 Functions

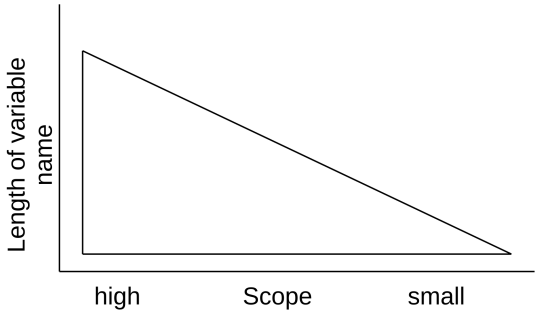
- Use verb or verb phrase names for functions.
- Keep functions as small as possible.
- Functions should do one thing and do it well - follow the Single Responsibility Principle (SRP).
- Use one level of abstraction per function.
- Avoid code duplication (Don't Repeat Yourself - DRY).
- Functions should have no side effects.
- Group similar functions together.
- Place functions in a downward direction, with higher-level functions calling lower-level ones.
- Use consistent and clear terminology for concepts.
- Minimize the number of function arguments, ideally zero or one.
- Avoid using flag arguments - split methods into independent methods without flags.
- `private` functions can have longer and more descriptive names and can be refined over time using the Boy Scout Rule.
- The visibility of a function determines the length of its name - shorter names for public functions, longer names for private ones.



Method name length

3.3.3 Variables

- Declare variables close to their usage and minimize their scope.
- The scope of a variable determines its name's length - longer names for global variables, shorter names for scope-limited variables like loop index variables.



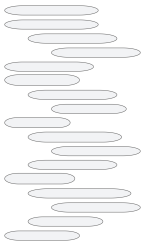
Variable name length

3.4 Shapes of code

Who doesn't know it, you sit back, look at the code from a distance and recognize patterns. These patterns tell you a lot about the code you are looking at. The outlines, peaks, valleys, and spaces between the lines of code can be used to detect certain antipatterns and improve the code.

So lean back, take a few steps back or set the screen resolution way too high and be excited about what you discover.

3.4.1 Spikes



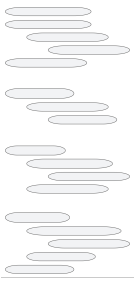
Advantage for understanding:

Each spike can first be analyzed on its own, even though it may be interdependent.

Advantage for refactoring:

Each spike is a potential candidate for transferring code into a separate private function and replacing it by calling that function.

3.4.2 Paragraphs



Advantage for understanding:

You know that the algorithm works in steps, and you know where the steps are in the code.

Advantage for refactoring:

Since steps should be somewhat distinct from each other, each step is a good candidate for outsourcing its code to a function. The resulting code would be a sequence of function calls. This would increase the level of abstraction and make the code more expressive.

3.4.3 Paragraphs with headers

Advantage for understanding:

As with paragraphs. The developer who wrote this has made your task easier by adding information to each step.

Advantage for refactoring:

As with paragraphs. You can use some terms in the comments as inspiration for function names. After refactoring, the comments become redundant and can be removed.

3.4.4 Suspicious comments

Advantage for understanding:

Not all comments are an advantage, and the code is often not a good one.

Advantage for refactoring:

Use the terms in the comments to rename the function and its parameters and remove the comments.

3.4.5 Intensive use of an object

Advantages for understanding:

The role of this code section is to set up this object.

Advantages for refactoring:

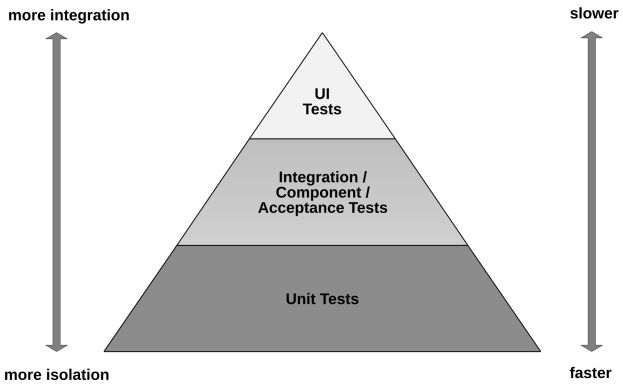
The function has several responsibilities, and one of them is to work with this object. Refactor this responsibility in a function to reduce the number of responsibilities (ideally to one) of the main function.

More information can be found here:
https://www.fluentcpp.com/2020/01/14/the-shapes-of-code/

4. Software Quality Assurance

4.1 Test Pyramid

Mike Cohn introduced the Test Pyramid in his book *Succeeding with Agile*. It is a great way to visualize different levels of testing. It also tells you how many tests to do on each level. The original test pyramid consists of three layers that your test suite should consist of *Unit Tests*, *Service Tests* and *User Interface Tests*.



The Test Pyramid

According to this pyramid, write many small and fast unit tests. Write some more coarse-grained integration tests and very few high-level tests that test your application from top to bottom.

Be careful not to end up with a Testing Pyramids & Ice-Cream Cones[1] that is a complete waste of time and maintenance.

Unit Tests
: Unit tests focus on a single class and are the easiest, cheapest, and fastest to complete. Unit tests are written at an early stage so that we get immediate feedback and know exactly where the bugs are.

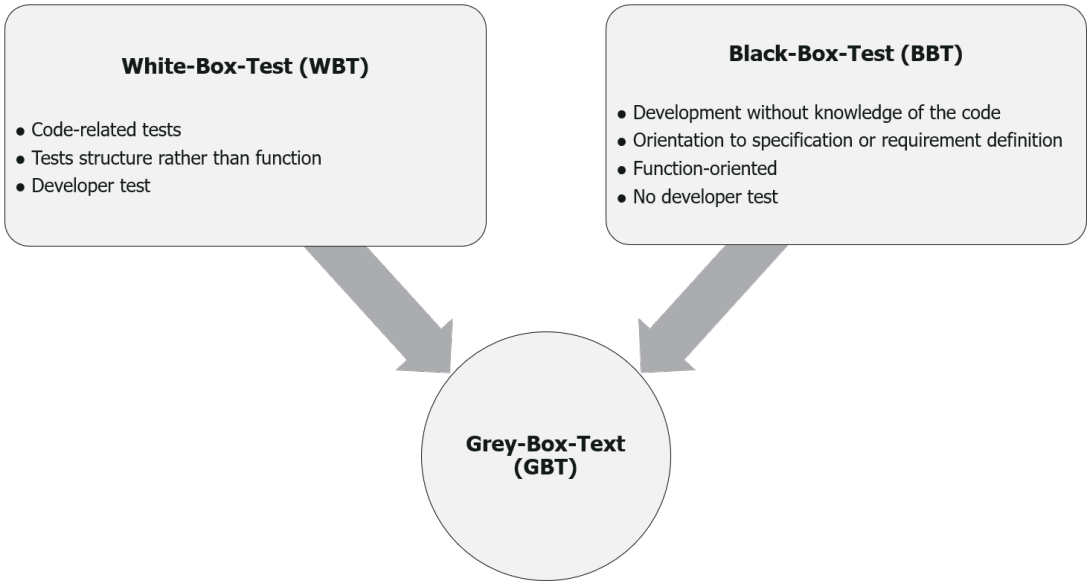
Integration / Component / Acceptance Tests
: Integration tests focus on the proper integration of different classes and modules. Compared to unit tests, they may require more specialized tools either for preparing the test environment or for interaction.

UI Tests
: UI tests focus on the verification of a system from the client‘s point of view. The entire application is tested in a real scenario, such as communication with the database, network, hardware, and other applications.

[1] https://alisterbscott.com/kb/testing-pyramids/

4.2 Test Classification

Tests can be classified according to their information status. There are three main testing strategies *White-Box-Test*, *Black-Box-Test*, and *Grey-Box-Test*. WBT is performed by the development team after coding is complete by testing the internal behavior of the code. BBT is performed by the professional test engineer by looking at the application. They do not have access to the logic and the flow of the code. GBT is the combination of WBT and BBT and is done without the knowledge of the implementation.



Differences between WBT, GBT and BBT

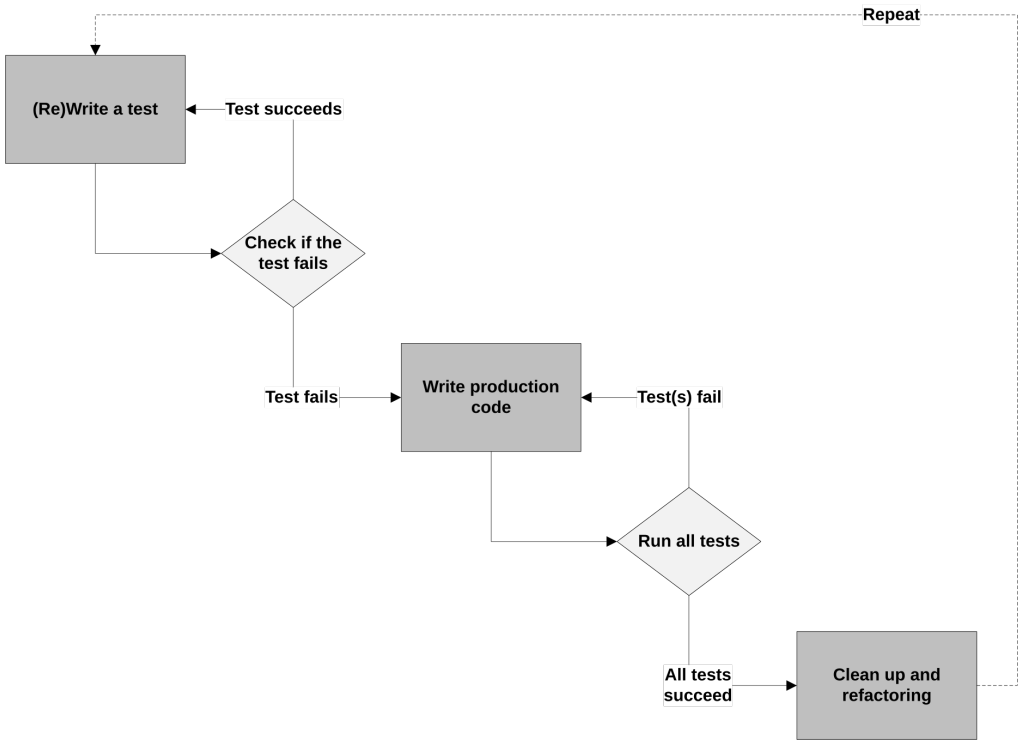
	WBT	BBT
Advantage	Testing of subcomponents and internal functionality	Better verification of the overall system
	Less organizational effort	Testing of semantic properties with suitable specification
	Automation through good tool support	Portability of systematically created test sequences to platform-independent implementations
Disadvantage	Compliance with a specification not verified	Greater organizational effort
	Possible testing *around errors*	Additional functions added during implementation are only tested by chance
		Test sequences of an insufficient specification are unusable

4.3 Test-driven Development (TDD)

When we write a test... We are telling ourselves a story about how the operation will look from the outside – Kent Beck

What is test-driven Development?

- Test-driven programming
  - Motivate any behavioral change to the code through an automated test.
- Refactoring
  - Keep the code in *simple form* and develop the design step by step during the programming.
- Frequent integration
  - Integrate the code as often as necessary.



Test-driven Development Workflow

Why test-driven development?

- TDD ensures the quality and maintainability of software.
- Tests ensure that the existing functions are retained when extending and revising.
- Refactoring extends the productive life of the software.
- Code is often difficult to test afterward.
- Test-First emphasizes the user view.
- Software development without testing is like climbing without rope and hooks.

Why does Test-driven development improve the quality?

If you can't write a test for something you don't understand it

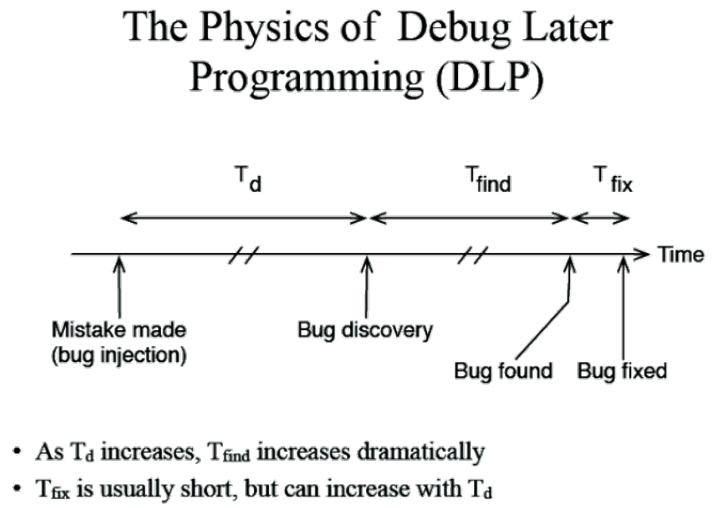
The practice of writing tests before implementation forces to deal more intensively with the problem to be solved and thus leads to a higher quality of the code, which is also more comprehensible for other developers. As a side-effect, TDD is also forced to deal with the question of what customers expect from the software to be written. TDD is therefore basically not a test method but a design method.

Without a regression test you can't clean the code

Without the possibility of quick testing, you often do not risk changing existing code. It is better to copy the code of a class, for example, into a new class, which is then extended with additional features. The result is an overloaded, confusing project with large amounts of superfluous code.

Fast feedback cycles save time and money

If you write the test first, the correctness of the implementation of a feature can be checked immediately. As a result, you always have certainty about when you have finished developing a feature.



Debug Later Programming

Without these rapid feedback cycles, you often get lost in the search for the causes of a bug. TDD helps to break down a big problem into smaller, manageable, and testable problems.

You are going to test it anyway, spend the time to do it right

Regression tests are indispensable anyway. Without performing them regularly, testing large applications becomes almost impossible. Of course, regression tests can also be written afterward, but then one does not benefit from the better design quality provided by TDD. Secondly, time tends to become limited as a deadline comes, so subsequent testing often comes up short.

It makes my work more enjoyable

Every successful test gives you the feeling of having created something valuable: a piece of executable software that you can proudly show to other developers.

4.4 Unit testing with JUnit 5

4.4.1 Unit Tests

Benefits of unit tests:

- Changes to the code can be checked immediately
- Better code quality/stability or security
- Less hesitation before making changes to core components
- Shorter development times, despite additional tests to be implemented
- Easier refactoring
- Simplified documentation of the implementation

Requirements for unit tests:

- Easy implementation of the tests
- Fast and automated test execution
- Thorough analysis of the test results

4.4.2 JUnit 5

JUnit 5[2] is a simple Java unit testing framework helping testing on the JVM with a focus on Java 8 language features, extensibility, and a modern programming API.

What are the goals of JUnit 5?

- Reduce test creation effort to the absolute minimum
- Easy to learn and use
- Avoid redundant work
- Tests must
  - be repeatedly applicable
  - can be produced separately
  - be incremental
  - can be freely combined
  - be feasible also by others than the author
  - can also be evaluated by others than the author
- Enable the use of test data
  - Reusability of test data
  - Test data generation is usually more complex than the test itself

[2] https://junit.org/junit5/

What should we test?

The best thing about unit tests is that you can write them for all your production code classes, regardless of their functionality, complexity, or internal structure. You can perform unit tests for controllers, services, repositories, domain classes, or utility classes. Follow the rule of one test class per production class. This is a good start, but not a must. If you feel that you need to split your test into several test classes, then it's fine, do that.

A unit test class should test the public interface of the class. If you do this, all other methods will be tested automatically in an indirect way.

If the package structure of your test class is the same as the production class, `protected` and `package-private` methods are accessible from a test class and can be tested as well but testing these methods may be too far. Private methods cannot be tested anyway, because you cannot call them from another test class.

Your test suite should ensure that all your non-trivial code paths are tested. At the same time, they should not be too closely bound to your implementation.

How to structure tests?

All structure technics has the following steps in common:

- Setting up the test data
- Calling the method under test
- Assert that the expected results are returned

There are two common patterns, which can be applied to your test structure like Arrange, Act, Assert[3] and Given, When, Then[4], which was developed as part of Behavior-Driven Development[5] (BDD).

What about project structure and naming conventions?

Nowadays all developers use standard tools for building their projects like Maven[6] or Gradle[7]. Thus, all production source code resides in `src/main/java` and all test source files are in `src/test/java`.

Typical project directory structure

```
└── src
    └── main
        ├── java
        │   └── HelloWorld.java
        test
        └── java
            └── HelloWorldTest.java
```

Test class names like the above usually will be prefixed with `Test`. This is a very common naming convention and should be followed consistently. It enables the developer to understand at once which class is being tested. Furthermore, some tools rely on convention.

[3] https://xp123.com/articles/3a-arrange-act-assert/
[4] https://martinfowler.com/bliki/GivenWhenThen.html
[5] https://dannorth.net/introducing-bdd/
[6] https://maven.apache.org/
[7] https://gradle.org/

4.4.3 First unit test

First simple JUnit 5 test

```
class SimpleTest {

    private Collection<String> collection = new ArrayList<>();

    @Test
    void testEmptyCollection() {
        assertTrue(collection.isEmpty());
    }

    @Test
    void testOneItemCollection() {
        collection.add("First JUnit Test");

        assertEquals(1, collection.size());
    }
}
```

Execution order

```
testEmptyCollection()
testOneItemCollection()
```

Execution order[8] of tests:

By default, test methods will be ordered using an algorithm that is deterministic but intentionally nonobvious. This ensures that subsequent runs of a test suite execute test methods in the same order, thereby allowing for repeatable builds. Even if unit tests should normally not depend on the order in which they are executed, there are times when it is necessary to enforce a certain order of execution of the test method. To specify the order in which test methods are executed, annotate your test class or test interface with `@TestMethodOrder` and provide the preferred `MethodOrder` implementation.

With JUnit 5 you can reduce the visibility of the test classes and methods. With JUnit 4 the classes and methods had to be `public`. In JUnit 5 test classes can have any visibility, it is recommended to use the default package visibility, which improves the test codebase.

[8] https://junit.org/junit5/docs/current/user-guide/#writing-tests-test-execution-order

4.4.4 Assertions

Assertions[9] is a collection of utility methods that support asserting conditions in tests. JUnit 5 contains many of the assertions of JUnit 4 and several interesting new assertions.

It also provides support for lambda expressions that can be used in assertions. The advantage of using lambda expressions for the assertion message is that it is evaluated lazily, which can reduce time and resources by avoiding the construction of complex messages.

The assertion result of a test provides a specific value. A failed assertion will throw an `AssertionFailedError` or subclass thereof.

4.4.4.1 `assertEquals()` / `assertArrayEquals()`

Assert that expected and actual are equal.

Equality

```
@Test
void equality() {
    Calculator calculator = new Calculator();

    assertEquals(2, calculator.add(1, 1));
    assertEquals(4, calculator.multiply(2, 2), "Optional failure message");

    char[] expected = {'J','u','n','i','t'};
    char[] actual = "Junit".toCharArray();

    assertArrayEquals(expected, actual);
}
```

4.4.4.2 `assertSame()` / `assertNotSame()`

Assert that expected and actual refer to the same object.

Identity

```
@Test
void identity() {
    User john = new User(1, "John", "Rambo");
    User rocky = new User(2, "Rocky", "Balboa");

    assertSame(john, john);
    assertNotSame(john, rocky);
}
```

[9] https://junit.org/junit5/docs/current/api/org.junit.jupiter.api/org/junit/jupiter/api/Assertions.html

4.4.4.3 `assertTrue()` / `assertFalse()` / `assertAll()`

Assert that the supplied condition is true or false. `assertAll()` assert that all supplied executables do not throw exceptions. It is a possibility to group assertions and have all the failed assertions reported together.

Truth

```
@Test
void truth() {
    assertTrue('a' < 'b', () -> "Assertion messages can be lazily evaluated"
            + "to avoid constructing complex messages unnecessarily.");
}

@Test
void groupedTruth() {
    Person person = new Person("Jane", "Doe");

    assertAll("person",
        () -> assertEquals("Jane", person.firstName()),
        () -> assertEquals("Doe", person.lastName())
    );
}
```

4.4.4.4 `assertNull()` / `assertNotNull()`

Assert that actual is `null` or not.

Existence

```
@Test
void existence() {
    assertNotNull(new Calculator());
    assertNull(null);
}
```

4.4.4.5 `assertThrows()`

Assert that execution of the supplied executable throws an exception of the expected type and return the exception.

Exceptions

```
@Test
void exceptions() {
    Calculator calculator = new Calculator();

    Exception exception = assertThrows(ArithmeticException.class, () -> calculator.divide(1, 0));
    assertEquals("/ by zero", exception.getMessage());
}
```

4.4.4.6 `assertTimeout()`

Assert that execution of the supplied executable completes before the given timeout is exceeded. This can be useful for testing the performance and efficiency of code, and for ensuring that it does not take an excessive amount of time to execute.

Timeout

```
@Test
void timeout() {
    assertTimeout(Duration.ofSeconds(5), () -> TimeUnit.SECONDS.sleep(1));
}
```

4.4.4.7 `fail()`

Is used to intentionally cause a test to fail.

Failing

```
@Test
void failing() {
    fail("a failing test");
}
```

4.4.5 Annotations

JUnit supports the following annotations[10] for configuring tests:

`@BeforeAll`
: Methods that are executed when the execution definition is started, before all tests. Annotated method must be `static`.

`@AfterAll`
: Methods that are executed after all tests, before closing the execution definition. For example, resources can be released here. Annotated method must be `static`.

`@BeforeEach`
: Methods that are performed before each test.

`@AfterEach`
: Methods that are performed after each test.

`@Test`
: The actual test methods. Only methods with this annotation are executed as tests.

`@Disabled`
: Temporary deactivation of test methods.

`@Tag` Used to declare tags for filtering, either at class or method level. Like Categories[11] in JUnit 4.

`@DisplayName`
: Specification of user-defined names for test classes and methods.

`@Timeout`
: Is used to fail a test if its execution exceeds a certain duration.

`@Nested`
: Is used to signal that the annotated class is nested. The nested test class must be an inner class, meaning a non-static class. The nested class can share the setup and state with an instance of its enclosing class. And, since inner classes cannot have static fields and methods, this prohibits the use of the `@BeforeAll` and `@AfterAll` annotations in nested tests.

`@ParameterizedTest`
: Is used to signal that the annotated method is a parameterized test method. You have to pick up at least one source of arguments. There are several types of parameter sources you can pick from.

`@TempDir`
: Can be used to annotate a non-private field in a test class or a parameter in a lifecycle method or test method of type `Path` or `File` that should be resolved into a temporary directory.

[10] https://junit.org/junit5/docs/current/user-guide/#writing-tests-annotations
[11] https://github.com/junit-team/junit4/wiki/Categories

4.4.5.1 `@Test`

- `@Test`[12] marks the methods to be tested.
- The annotated methods must not be `private` or `static`.
- Must not return a value, thus the method should return `void`.
- Only with `@Test` annotated methods will be executed.

@Test

```
class TestAnnotationTest {

    @Test
    void simple() {
        Collection<Object> collection = new ArrayList<>();
        assertTrue(collection.isEmpty());
    }

    @Test
    void lambdaExpressions() {
        assertTrue(Stream.of(1, 2, 3)
          .stream()
          .mapToInt(i -> i)
          .sum() > 5, () -> "Sum should be greater than 5");
    }

    @Test
    void groupAssertions() {
        int[] numbers = {0, 1, 2, 3, 4};

        assertAll("numbers",
            () -> assertEquals(numbers[0], 1),
            () -> assertEquals(numbers[3], 3),
            () -> assertEquals(numbers[4], 1)
        );
    }
}
```

[12] https://junit.org/junit5/docs/current/api/org.junit.jupiter.api/org/junit/jupiter/api/Test.html

4.4.5.2 `@BeforeEach / @AfterEach`

- `@BeforeEach`[13] is used to specify that the annotated method should be executed before **each** test in the current test class.
- `@AfterEach`[14] is used to specify that the annotated method should be executed after **each** test in the current test class.

@BeforeEach / @AfterEach

```java
class BeforeAndAfterEachAnnotationsTest {
    private Collection<String> collection;

    @BeforeEach
    void setUp() {
        collection = new ArrayList<>();
    }

    @AfterEach
    void tearDown() {
        collection.clear();
    }

    @Test
    void testEmptyCollection() {
        assertTrue(collection.isEmpty());
    }

    @Test
    void testOneItemCollection() {
        collection.add("itemA");
        assertEquals(1, collection.size());
    }
}
```

Execution order

```
setUp()
testEmptyCollection()
tearDown()

setUp()
testOneItemCollection()
tearDown()
```

[13] https://junit.org/junit5/docs/current/api/org.junit.jupiter.api/org/junit/jupiter/api/BeforeEach.html
[14] https://junit.org/junit5/docs/current/api/org.junit.jupiter.api/org/junit/jupiter/api/AfterEach.html

4.4.5.3 `@BeforeAll / @AfterAll`

- `@BeforeAll`[15] is used to specify that the annotated method should be executed before **all** tests in the current test class.
- `@AfterAll`[16] is used to specify that the annotated method should be executed after **all** tests in the current test class.
- These annotated methods must be `static void` and are only executed **once** for a given test class.

@BeforeAll / @AfterAll

```java
class BeforeAndAfterAllAnnotationsTest {
    private Collection<String> collection;

    @BeforeAll
    static void oneTimeSetUp() {}
    @AfterAll
    static void oneTimeTearDown() {}

    @BeforeEach
    void setUp() {
        collection = new ArrayList<>();
    }
    @AfterEach
    void tearDown() {
        collection.clear();
    }

    @Test
    void testEmptyCollection() {
        assertTrue(collection.isEmpty());
    }
    @Test
    void testOneItemCollection() {
        collection.add("itemA");
        assertEquals(1, collection.size());
    }
}
```

Execution order

```
oneTimeSetUp()

setUp()
testEmptyCollection()
tearDown()

setUp()
testOneItemCollection()
tearDown()

oneTimeTearDown()
```

[15] https://junit.org/junit5/docs/current/api/org.junit.jupiter.api/org/junit/jupiter/api/BeforeAll.html
[16] https://junit.org/junit5/docs/current/api/org.junit.jupiter.api/org/junit/jupiter/api/AfterAll.html

4.4.5.4 `@Disabled`

- `@Disabled`[17] is used to specify that the annotated test class or test method is currently disabled and should not be executed.
- A reason can be declared to the annotation as well to provide more information why the test class or test method is disabled.
- Can be applied at the class level as well, which disables **all** test methods within that class.

Exclusion of test methods

```
1  class DisabledAnnotationTest {
2
3      @Disabled
4      @Test
5      void disabled() {
6      }
7  }
```

Exclusion of test methods with comment

```
1  class DisabledAnnotationTest {
2
3      @Disabled("not ready yet")
4      @Test
5      void disabledWithComment() {
6      }
7  }
```

Exclusion of test classes

```
1  @Disabled
2  class DisabledAnnotationTest {
3
4      @Test
5      void disabled() {
6      }
7
8      @Test
9      void disabledAsWell() {
10     }
11 }
```

[17] https://junit.org/junit5/docs/current/api/org.junit.jupiter.api/org/junit/jupiter/api/Disabled.html

4.4.5.5 `@DisplayName`

- `@DisplayName`[18] is used to declare a custom display name for the annotated test class or test method.
- Display names are used for reporting in IDEs and build tools and may contain spaces, special characters, and even emoji.

@DisplayName

```
1  @DisplayName("A special test case")
2  class DisplayNameAnnotationTest {
3
4      @Test
5      @DisplayName("Custom test name containing spaces")
6      void testWithDisplayNameContainingSpaces() {
7      }
8
9      @Test
10     @DisplayName("°□°")
11     void testWithDisplayNameContainingSpecialCharacters() {
12     }
13
14     @Test
15     @DisplayName("□")
16     void testWithDisplayNameContainingEmoji() {
17     }
18 }
```

4.4.5.6 `@Tag`

- `@Tag`[19] marks test classes or test methods.
- These tags can later be used to filter the recognition and execution of tests.

@Tag

```
1  @Tag("fast")
2  @Tag("smoke-test")
3  class TagAnnotationTest {
4
5      @Test
6      @Tag("math")
7      void testingMathCalculation() {
8      }
9  }
```

[18] https://junit.org/junit5/docs/current/api/org.junit.jupiter.api/org/junit/jupiter/api/DisplayName.html
[19] https://junit.org/junit5/docs/current/api/org.junit.jupiter.api/org/junit/jupiter/api/Tag.html

4.4.5.7 `@Timeout`

- `@Timeout`[20] is used to define a timeout for a method or all testable methods within one class and its `@Nested` classes.
- Applying this annotation to a test class has the same effect as applying it to all testable methods.

@Timeout

```
class TimeoutAnnotationTest {

    @BeforeEach
    @Timeout(5)
    void setUp() {
        // fails if execution time exceeds 5 seconds
    }

    @Test
    @Timeout(value = 100, unit = TimeUnit.MILLISECONDS)
    void failsIfExecutionTimeExceeds100Milliseconds() {
        // fails if execution time exceeds 100 milliseconds
    }

}
```

[20] https://junit.org/junit5/docs/current/api/org.junit.jupiter.api/org/junit/jupiter/api/Timeout.html

4.4.6 Assumptions

Assumptions[21] are used to perform tests only if certain conditions are met. This is typically used for external conditions that are necessary for the test to run properly. If the condition does not apply, the method is exited.

4.4.6.1 `assumeFalse()`

Is used to skip a test if a given condition is false.

assumeFalse()

```
@Test
void falseAssumption() {
    assumeFalse(5 < 1);

    assertEquals(7, 5 + 2);
}
```

4.4.6.2 `assumeTrue()`

Is used to skip a test if a given condition is true.

assumeTrue()

```
@Test
void trueAssumption() {
    assumeTrue(5 > 1);

    assertEquals(7, 5 + 2);
}
```

4.4.6.3 `assumingThat()`

It can be used to skip a test if certain conditions are not met.

assumingThat()

```
@Test
void assumptionThat() {
    String message = "Just a assumptionThat test";

    assumingThat(
        "Just a assumptionThat test".equals(message),
        () -> assertEquals(2 + 2, 4)
    );
}
```

[21] https://junit.org/junit5/docs/current/api/org.junit.jupiter.api/org/junit/jupiter/api/Assumptions.html

4.4.7 Parameterized Tests

Parameterised tests[22] make it possible to run a test several times with different arguments. They are declared just like normal `@Test` methods but use the `@ParameterizedTest` annotation instead.

JUnit supports different sources as arguments:

- `@ValueSource`
- `@EnumSource`
- `@MethodSource`
- `@CsvSource`
- `@CsvFileSource`
- `@ArgumentsSource`
- `@EnumSource`
- `@NullSource`
- `@EmptySource`
- `@NullAndEmptySource`

4.4.7.1 `@ValueSource`

- `@ValueSource`[23] provides access to an array of values.
- Supported types are `shorts`, `bytes`, `ints`, `longs`, `floats`, `doubles`, `chars`, `booleans`, `strings` and `classes`.
- Only one of the supported types may be specified per `@ValueSource` declaration.

@ValueSource

```
1  @ParameterizedTest
2  @ValueSource(strings = { "otto", "level", "radar", "rotor", "kayak" })
3  void palindromes(String candidate) {
4      assertTrue(StringUtils.isPalindrome(candidate));
5  }
6
7  @ParameterizedTest
8  @ValueSource(ints = { 1, 2, 3 })
9  void testWithValueSource(int argument) {
10     assertTrue(argument > 0 && argument < 4);
11 }
```

[22] https://junit.org/junit5/docs/current/user-guide/#writing-tests-parameterized-tests
[23] https://junit.org/junit5/docs/current/api/org.junit.jupiter.params/org/junit/jupiter/params/provider/ValueSource.html

4.4.7.2 `@MethodSource`

- `@MethodSource`[24] provides access to values returned from factory methods of the class in which this annotation is declared or from static factory methods in external classes referenced by fully qualified method name.
- Each factory method must generate a stream of arguments.
- A stream is anything that JUnit can convert into a `Stream`, such as `Stream`, `DoubleStream`, `LongStream`, `IntStream`, `Collection`, `Iterator`, `Iterable`, an array of objects, or an array of primitives.
- Factory methods must be `static` and not declare any parameters.

@MethodSource - Stream<String>

```java
@ParameterizedTest
@MethodSource("stringProvider")
void testWithExplicitLocalMethodSource(String argument) {
    assertNotNull(argument);
}

static Stream<String> stringProvider() {
    return Stream.of("JUnit 5", "rocks!");
}
```

@MethodSource - IntStream

```java
@ParameterizedTest
@MethodSource("range")
void testWithRangeMethodSource(int argument) {
    assertNotEquals(9, argument);
}

static IntStream range() {
    return IntStream.range(0, 20).skip(10);
}
```

@MethodSource - Stream<Arguments>

```java
@ParameterizedTest
@MethodSource("stringIntAndListProvider")
void testWithMultiArgMethodSource(String str, int num, List<String> list) {
    assertEquals(5, str.length());
    assertTrue(num >=1 && num <=2);
    assertEquals(2, list.size());
}

static Stream<Arguments> stringIntAndListProvider() {
    return Stream.of(
        arguments("apple", 1, Arrays.asList("a", "b")),
        arguments("lemon", 2, Arrays.asList("x", "y"))
    );
}
```

[24] https://junit.org/junit5/docs/current/api/org.junit.jupiter.params/org/junit/jupiter/params/provider/MethodSource.html

4.4.7.3 `@CsvSource`

- `@CsvSource`[25] reads comma-separated values (CSV) from one or more supplied CSV lines.

@CsvSource

```
@ParameterizedTest
@CsvSource({
    "Bischofsmais,         1",
    "Osternohe,            2",
    "'Leogang, Austria', 0xF1",
    "'Porte du Soleil, France',        4",
})
void testWithCsvSource(String bikepark, int rank) {
    assertNotNull(bikepark);
    assertNotEquals(0, rank);
}
```

4.4.7.4 `@CsvFileSource`

@CsvFileSource

```
@ParameterizedTest
@CsvFileSource(resources = "/two-column.csv", numLinesToSkip = 1)
void testWithCsvFileSource(String country, int reference) {
    assertNotNull(country);
    assertNotEquals(0, reference);
}
```

@CsvFileSource

```
#two-column.csv
Country, reference
Sweden, 1
Germany, 2
Australia, 3
"United States of America", 4
```

[25] https://junit.org/junit5/docs/current/api/org.junit.jupiter.params/org/junit/jupiter/params/provider/CsvSource.html

4.4.7.5 `@ArgumentsSource`

@ArgumentsSource

```java
@ParameterizedTest
@ArgumentsSource(BikeArgumentsProvider.class)
void testWithArgumentsSource(String argument) {
    assertNotNull(argument);
}
```

ArgumentsProvider

```java
public class BikeArgumentsProvider implements ArgumentsProvider {

    @Override
    public Stream<? extends Arguments> provideArguments(ExtensionContext context) {
        return Stream.of("Specialized", "Cube").map(Arguments::of);
    }
}
```

4.5 More on Unit Tests

4.5.1 Heuristics

Testing features, not methods

- Test cases are based on requirements and features, not on the methods to be tested.
- There is no 1:1 correspondence between the test case method and the tested method.
- Test cases should document how to use the classes correctly using examples.

Bad code

```
class EuroTest {
    @Test void newEuro() {...}
    @Test void getIntAmount() {...}
    @Test void getAmount() {...}
    @Test void plus() {...}
    @Test void minus() {...}
}
```

Good code

```
class EuroTest {
    @Test void createFromInt() {...}
    @Test void createFromString() {...}
    @Test void rounding() {...}
    @Test void simpleAddition() {...}
}
```

Testing at the edges

- Most algorithmic errors occur at the edges of the allowed value ranges.
- Can the value range be divided into several samples equivalent for the test? At least one sample per equivalence class!
- What is not tested may not work.

Implementation independence

- Tests are directed against the public class interface.
- Tests based on the innards of a class are extremely fragile.
- Wanting access to variables or private methods shows that the code still lacks a crucial design idea.

Orthogonal test cases

- Test cases are independent of each other if they refer to orthogonal aspects.

- Often a test can be extremely simplified by making assumptions that another test has already verified.
- If one gets into the trouble of having to adapt too many tests just to make a code change, the tests are not orthogonal.

Record results in the test

- Expected values are coded as constants, not calculated again in the test.
- If we reproduce application logic in the test, we also reproduce its errors.

Performance

Bad code

```
class EuroTest {

    @Test
    void multiYearInterest() {
        double amount = 100.0;
        double interest = 5.0;
        double expectedInterest = amount * Math.pow((1 + interest / 100.0), 3.0);

        assertEquals(expectedInterest, calculator.interest(amount, interest, 3), 0.001);
    }
}
```

Good code

```
class EuroTest {

    @Test
    void multiYearInterest() {
        double amount = 100.0;
        double interest = 5.0;

        assertEquals(115.76, calculator.interest(amount, interest, 3), 0.001);
    }
}
```

Do not forget about exceptions and errors!

- Often the error cases are insufficiently tested. But exactly these are important to understand how the code behaves when it is not used correctly.

Remove redundancy in test code!

- The DRY Principle applies to test code as well.
- Redundancies should be avoided to reduce the impact on test code in the event of future changes to the production code.

Keep test cases short and understandable!

- Do not write tests with big test methods.
- Split the test into multiple test methods.

Choose meaningful test case names!

- Naming is hard but try to communicate with the method test name what the test is doing.
- Be consistent and choose a naming approach that fits best.
- Test features not methods, this helps to divide tests in clean test cases with mean fulling method names.

Do not trade test code as the second citizen in your codebase!

- Test code should be no less important than production code.
- Do not treat test code worse, treat it like production code and use the same coding standards.

Remove flaky tests!

- Since software tests serve as an early warning of potential regressions, they should always work reliably. A failed test should be cause for concern, and a broken build should immediately investigate why the test failed. It is a stop-the-world event.
- This approach can only work for tests that fail in a deterministic way. A test that sometimes fails and sometimes passes is unreliable and completely corrupts the entire test suite. This can have negative effects on the team regarding tests.
- Developers no longer trust tests and soon ignore them. Even if nonflaky tests fail, it is difficult to detect them in several broken tests. On the other hand, it is difficult to understand whether new failures are new or whether they come from existing flaky tests.

4.5.2 Naming of test methods

There are several ways to name the test methods. The most important thing is to be consistent and to define a convention within the team.

Approach 1:

Describe the facts of the test case.

Fact of the test as name

```
@Test void newAccount() {}
@Test void withdraw() {}
@Test void cannotWithdrawNegativeAmount() {}
@Test void cannotWithdrawUncoveredAmount() {}
```

Approach 2:

Describe the desired behavior of the test case.

Desired behaviour as name

```
@Test newAccountShouldReturnCustomer() {}
@Test newAccountShouldHaveZeroBalance() {}
@Test withdrawShouldReduceBalanceByAmount() {}
@Test withdrawNegativeAmountShouldThrowException() {}
@Test withdrawNegativeAmountShouldNotChangeBalance() {}
```

4.5.3 Object Mother

An Object Mother[26] is a class that can be used in testing, which allows us to provide pre-configured objects for our tests. This example here is rather trivial but imagine this in a real application with classes having many attributes and/or compositions.

Most tests no longer instantiate objects themselves but go through a factory. In a future evolution, if a new mandatory attribute is added to a class, then all you must do is to modify the factory and all the tests will pass.

The combination of builders allows the objects generated by the Object Mother to be customized for the needs of the test. Otherwise, it would be necessary to multiply the factories for each need with a method taking in parameter all the necessary information for the tested case.

Classical approach

```
User user = new User("user", "password", "USER")
Authentication auth = new UsernamePasswordAuthenticationToken(user, null, user.authorities());
```

Object Mother Pattern

```
Authentication user = TestAuthentications.authenticatedUser();
Authentication admin = TestAuthentications.authenticatedAdmin();
Authentication tester = TestAuthentications.authenticated(new User("user", "password", "TESTER"));
```

Object Mother Pattern implementation

```
public final class TestAuthentications {

    private static final User USER = new User("user", "password", "USER");
    private static final User ADMIN = new User("admin", "password", "USER, ADMIN");

    private TestAuthentications() {
    }

    public static Authentication authenticatedAdmin() {
        return authenticated(ADMIN);
    }

    public static Authentication authenticatedUser() {
        return authenticated(USER);
    }

    public static Authentication authenticated(User user) {
        return new UsernamePasswordAuthenticationToken(user, null, user.authorities());
    }
}
```

[26] https://martinfowler.com/bliki/ObjectMother.html

4.5.4 Test Data Builder

The Object Mother and Builder patterns each bring their own set of advantages to our tests but can also be combined to further enhance the readability and maintainability of our tests. For a complex object the Test Data Builder Pattern described in the book *Growing Object-Oriented Software, Guided by Tests* is a perfect solution for creating test objects.

Use of a Test Data Builder

```
Invoice anInvoice = new InvoiceBuilder().build();
```

Builder Pattern

```
public class InvoiceBuilder {

    Recipient recipient = new RecipientBuilder().build();
    InvoiceLines lines = new InvoiceLines(new InvoiceLineBuilder().build());
    PoundsShillingsPence discount = PoundsShillingsPence.ZERO;

    public InvoiceBuilder withRecipient(Recipient recipient) {
        this.recipient = recipient;
        return this;
    }
    public InvoiceBuilder withInvoiceLines(InvoiceLines lines) {
        this.lines = lines;
        return this;
    }
    public InvoiceBuilder withDiscount(PoundsShillingsPence discount) {
        this.discount = discount;
        return this;
    }
    public Invoice build() {
        return new Invoice(recipient, lines, discount);
    }
}
```

Combined Builder Pattern

```
 Invoice invoiceWithNoPostcode = new InvoiceBuilder()
    .withRecipient(new RecipientBuilder()
        .withAddress(new AddressBuilder()
            .withNoPostcode()
            .build())
        .build())
    .build();
```

Combined Builder Pattern, shorter

```
Invoice invoice = new InvoiceBuilder()
    .withRecipient(new RecipientBuilder().withAddress(new AddressBuilder().withNoPostcode()))
    .build();
```

Combined Builder Pattern, even shorter

```
Invoice invoice = anInvoice()
    .fromRecipient(aRecipient().withAddress(anAddress().withNoPostcode()))
    .build();
```

Combined Builder Pattern, shortest

```
Invoice invoice = anInvoice()
    .from(aRecipient().with(anAddress().withNoPostcode()))
    .build();
```

4.5.5 F.I.R.S.T

Follow five rules that help writing clean tests, which define the **F.I.R.S.T**[27] acronym.

Fast

- Tests should be fast.
- If tests are slow, they are run less frequently and therefore do not find problems early enough.

Independent

- Tests should not be interdependent but should be executable independently.
- A test should not set conditions for the next test and should be executable in any order.
- If tests are interdependent, the first one to fail triggers a cascade of further failed tests downstream.

Repeatable

- Tests should be repeatable in any environment.
- Tests should be executable in the production environment, in the QA environment, and on the laptop on the train without a network.

Self-Validating

- Tests should have a boolean output.
- Either they are passed, or they fail.
- The manual study of log files should be avoided.
- If the tests do not validate themselves, failure can become subjective and a long manual evaluation may be required to execute the tests.

Timely

- Tests must be written in time.
- Unit tests should be written just before the production code which ensures that they are passed.
- If tests are developed after the production code, the production code may be difficult to test.

[27] **Robert**, **Martin**: *Clean Code – A Handbook of Agile Software Craftsmanship*

4.6 Mocking with Mockito

Mockito[28] is a Java mocking framework for creating test double objects for unit tests of Java programs.

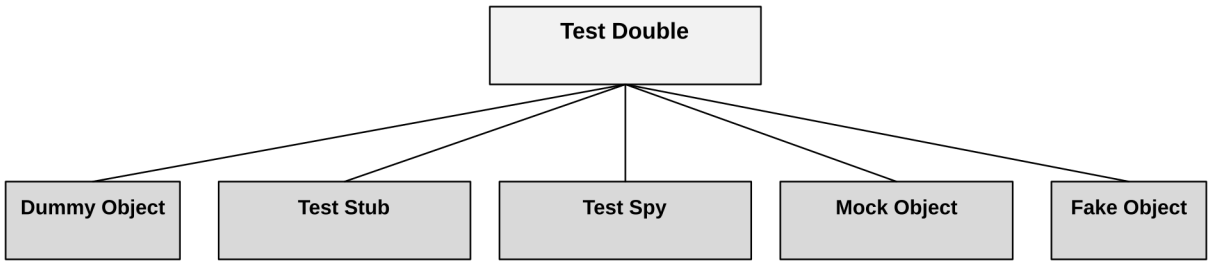
Why do we need test double objects?

Most of the time the code we write depends on other dependencies. If we are using unit testing, often the code delegates some work to other methods in other classes or our test must depend on those methods. But we want the tests to be independent of all other dependencies.

Test double objects help us to isolate a unit of code and test it alone. Replace DOCs (*Depended On Component*) and get control over the environment in which the SUT (*System Under Test*) is running and test the state and verify interactions.

4.6.1 Types of Test Double

Test Double is a generic term used by *Gerard Meszaros* in his book *xUnit Test Patterns* to mean any case where you replace a production object for testing purposes.



Types of Test Double

Test stub
: used for providing the tested code with *indirect input*.

Mock object
: used for verifying *indirect output* of the tested code, by first defining the expectations before the tested code is executed.

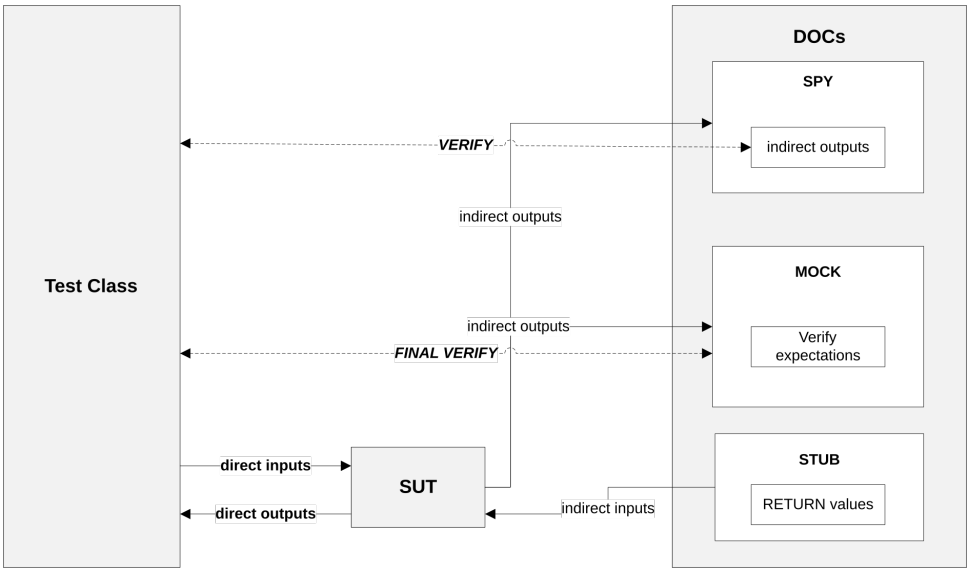
Test spy
: used for verifying *indirect output* of the tested code, by asserting the expectations afterward, without having defined the expectations before the tested code is executed. It helps in recording information about the indirect object created.

Fake object
: used as a simpler implementation, e.g. using an in-memory database in the tests instead of doing real database access.

Dummy object
: used when a parameter is needed for the tested method but without needing to use the parameter.

[28] https://site.mockito.org/

Dummies and stubs are used for **preparing the environment** for testing. Spies and mocks are for **verifying the correctness** of the communication.



Types of Test Double

There are two styles[29] regarding mocking, the classical and the mockist TDD style. The classic TDD style is to use real objects when possible and a double when it is uncomfortable to use the real thing. However, a mocking TDD practitioner will always use a mock.

4.6.2 Activation

Activating Mockito can be done with JUnit 5 `@ExtendWith` annotation or programmatically, by invoking `MockitoAnnotations.openMocks()`.

@ExtendWith

```
@ExtendWith(MockitoExtension.class)
class MockitoAnnotationTest {
  ...
}
```

MockitoAnnotations.openMocks()

```
class MockitoTest {

  @BeforeEach
  void init() {
      MockitoAnnotations.openMocks(this);
  }
}
```

[29] https://martinfowler.com/articles/mocksArentStubs.html#ClassicalAndMockistTesting

4.6.3 Annotations

Mockito supports the following annotations[30] for configuring mocked tests:

`@Mock`
: Will create a new mock implementation for the given class. It will not create a real object.

`@Spy`
: Create a wrapper around a real instance and spy on that real object.

`@InjectMocks`
: Will inject the created mock to a given class instance.

`@Captor`
: Is used to create an ArgumentCaptor[31] instance which is used to capture method argument values for further assertions.

4.6.3.1 `@Mock`

With `@Mock`, a mock instance of a class will be created.

- The instance is entirely instrumented to track interactions with it.
- This is not a real object and does not maintain the state changes.

Manually with `Mockito.mock()`

```
@Test
void whenNotUsingMockAnnotation() {
    List mock = Mockito.mock(ArrayList.class);

    mock.add("Java");
    Mockito.verify(mock).add("Java");
    assertEquals(0, mock.size());

    Mockito.when(mock.size()).thenReturn(2020);
    assertEquals(2020, mock.size());
}
```

`@Mock` Annotation

```
@Mock
List<String> mock;

@Test
void whenUsingMockAnnotation() {
    mock.add("Java");
    Mockito.verify(mock).add("Java");
    assertEquals(0, mock.size());

    Mockito.when(mock.size()).thenReturn(2020);
    assertEquals(2020, mock.size());
}
```

[30] https://javadoc.io/doc/org.mockito/mockito-core/latest/org/mockito/Mockito.html
[31] https://javadoc.io/doc/org.mockito/mockito-core/latest/org/mockito/ArgumentCaptor.html

4.6.3.2 `@Spy`

With `@Spy`, a real instance of the class will be created.

- You can track every interaction with it.
- It maintains the state changes.

Manually with `Mockito.spy()`

```
@Test
void whenNotUsingSpyAnnotation() {
    List<String> spy = Mockito.spy(new LinkedList<>());

    spy.add("Java");
    Mockito.verify(spy).add("Java");
    assertEquals(1, spy.size());

    Mockito.doReturn(2020).when(spy).size();
    assertEquals(2020, spy.size());
}
```

`@Spy` Annotation

```
@Spy
List<String> spy = new LinkedList<>();

@Test
void whenUsingSpyAnnotation() {
    spy.add("Java");
    Mockito.verify(spy).add("Java");
    assertEquals(1, spy.size());

    Mockito.doReturn(2020).when(spy).size();
    assertEquals(2020, spy.size());
}
```

4.6.3.3 `@Captor`

@Captor

```
@ExtendWith(MockitoExtension.class)
class MockitoCaptorTest {

    @Mock
    private List mock;

    @Captor
    private ArgumentCaptor captor;

    @Test
    void whenUseCaptorAnnotation() {
        mock.add("Java");
        Mockito.verify(mock).add(captor.capture());

        assertEquals("Java", captor.getValue());
    }
}
```

4.7 Code Coverage

Code coverage is a metric that measures at runtime which source code lines have been processed. The measurement is done by executing unit test cases and gives you a metric of the degree to which the source code has been tested.

This allows you to make a statement about how comprehensively the code has been tested. It improves the SW quality through better quality of the test cases and increases the efficiency of the test cases.

Test coverage should **never** be understood as metrics about the quality of the tests or the code! It is only an indicator for problem areas. Can therefore only make negative statements, not positive ones.

Measurement techniques:

Statement Coverage / Line Coverage

- Measures whether and how often a single line of code has been run through.
- Problem: If the line is a logical comparison, a single run is not representative.

Decision Coverage / Branch Coverage

- For case distinctions (if, while, etc.) it is additionally checked that both cases (true and false) have occurred.

Path Coverage

- Path Coverage measures whether all possible combinations of program flow paths have been run through.
- Problem: The number of possibilities increases exponentially with the number of decisions.

Function Coverage

- Measures whether they have been called based on the functions.

Race Coverage

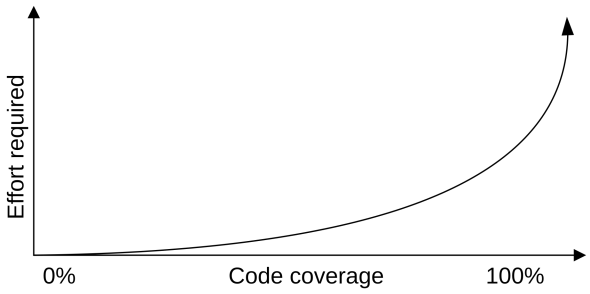
- Concentrates on code points that run in parallel.

Most code coverage tools on the market only support Statement and Decision Coverage, in some cases Path Coverage. Nevertheless, they already provide very valuable statements, e.g. code parts that were simply forgotten during testing and exceptions that have not been considered.

Challenges:

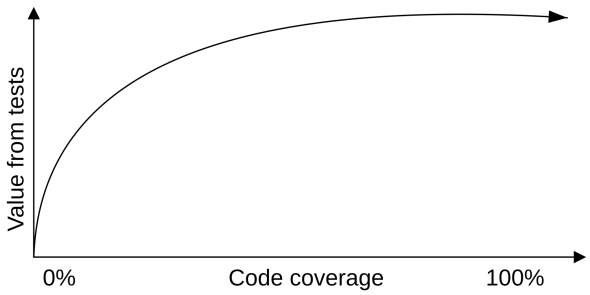
A 100% coverage is an illusion! Do not try to achieve 100% total code coverage. Achievable coverage depends strongly on the project type, whereas realistic values are in the range of 80%. Code coverage should be defined project-specifically and automatically checked during the build process.

Attaining 100% code coverage sounds good in theory but is almost always a waste of time. We will spend a lot of effort to get from 80% to 100%, as it is much more difficult than getting from 0% to 20%.



Effort required

Increasing code coverage has lower revenues as well. Don't waste your time trying to achieve a very high level of test coverage, just to be better positioned with your project. Often stupid tests are written to artificially push them up. It is better to have tests that you can trust and test the features properly. Testing pure setters and getters make not always sense. These tests only cost time and money and do not help the project.



Value from tests

Again, code coverage should not be used as a representation of the quality of a project!

Tooling:

Eclemma[32]

Java-based code coverage framework, with support of statement and decision coverage. Provide generation of text, HTML, and XML reports and is part of Eclipse.

[32] http://www.eclemma.org/

4.8 Static Code Analysis

Static code analysis is a static software test procedure. In contrast to dynamic test procedures, the software to be tested does not have to be executed.

Therefore, only the source code is needed. There is no need to run the system or a special test environment. It can be done by manual testing, but also automatically by a program. But should be anchored as early as possible in the development process.

Static code analysis is also used in compilers and IDEs for:

- Type check, *e.g. correct use of types*
- Style check, *e.g. use of code guidelines*
- Bug finding, *e.g. determining the use of uninitialized variables*

Tooling:

SonarQube[33]
: Platform for static code analysis of the technical quality. Supports not only Java but also the analysis of other programming languages. Analyses the code for different quality ranges and displays the results graphically.

Checkstyle[34]
: Checkstyle is an open-source program that supports the compliance of coding conventions in Java programs.

SpotBugs[35]
: Open source program that uses static code analysis to check Java programs for numerous bugs. Is the successor of FindBugs.

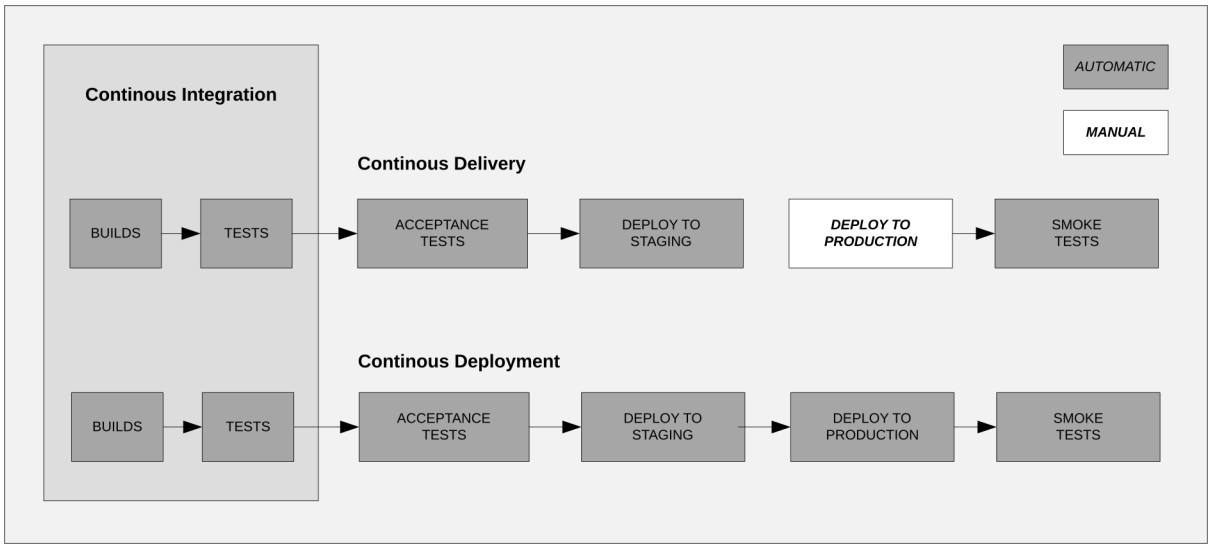
PMD[36]
: PMD is a static source code analyzer. It finds common programming flaws like unused variables, empty catch blocks, unnecessary object creation, and so forth. Include CPD, a copy-paste detector.

UCDetector[37]
: Unnecessary Code Detector searches for non-referenced methods and classes and advises on the visibility of methods and classes.

[33] http://www.sonarqube.org/
[34] http://checkstyle.sourceforge.net/
[35] https://spotbugs.github.io/
[36] https://pmd.github.io/
[37] http://www.ucdetector.org/

4.9 Continuous Integration

Continuous integration describes a process of regular, complete rebuilding, testing and distribution of an application. There are cloud providers like CircleCI[38] or Travis[39] or local servers like Jenkins[40], GitLab CI[41], TeamCity[42] and Bamboo[43].



Continuous Integration

Idea:

- Early and more frequent check-in of changes into the version management system.
- Replacement of large changes with incremental, functional small changes.
- When changes are checked into the version management system, the entire system is rebuilt and automatically tested.
- Try to give the developer feedback on his changes as soon as possible.
- Improve quality by replacing the traditional approach to testing with continuous testing and rapid feedback.

Parts and steps:

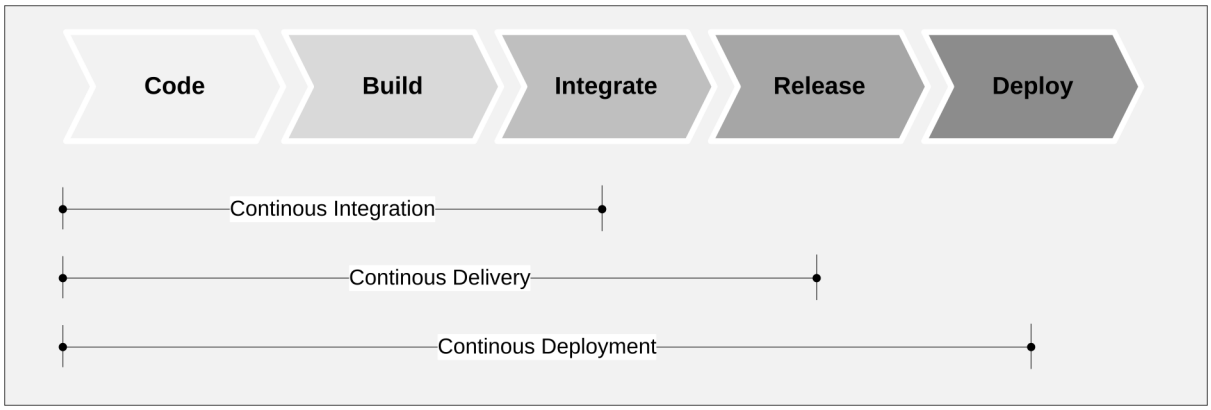
- Compilation
- Test execution (Unit tests, Integration tests, Acceptance tests, etc.)
- Integration (Database, Third-party systems)
- Static analysis (Code & Architecture)
- Automatic deployment
- Generation of documentation

[38] https://circleci.com/
[39] https://travis-ci.org/
[40] https://www.jenkins.io/
[41] https://docs.gitlab.com/ee/ci/
[42] https://www.jetbrains.com/teamcity/
[43] https://www.atlassian.com/software/bamboo

Addressed risks:

- Late troubleshooting
- Lack of team coordination
  - *Your changes don't match mine*
  - *Didn't you fix this two months ago?*
- Poor code quality
  - *Why do three different classes do the same thing?*
  - *The code of Team XY looks completely different*
- Lack of transparency/visibility
  - *What tests aren't running?*
  - *What does build 1.2.3 contain?*
  - *Where do we stand with code coverage?*
- Software not available
  - *I'm all right*
  - *It's working*
  - *I need another build to test*
  - *Tomorrow the boss is coming, we need a demo*

4.9.1 Differences between CI, CD, and CD



Differences between CI, CD and CD

Continuous Integration

- With CI, code changes are tested immediately after check-in, so that the newly included changes can be tested continuously.
- The basic idea is to test the code frequently during the development process to detect and fix potential problems early on.
- With CI, most of the work is done by automated tests and a build server.
- Since CI provides constant feedback on the software status, code problems that arise are usually less complex and much easier to solve.

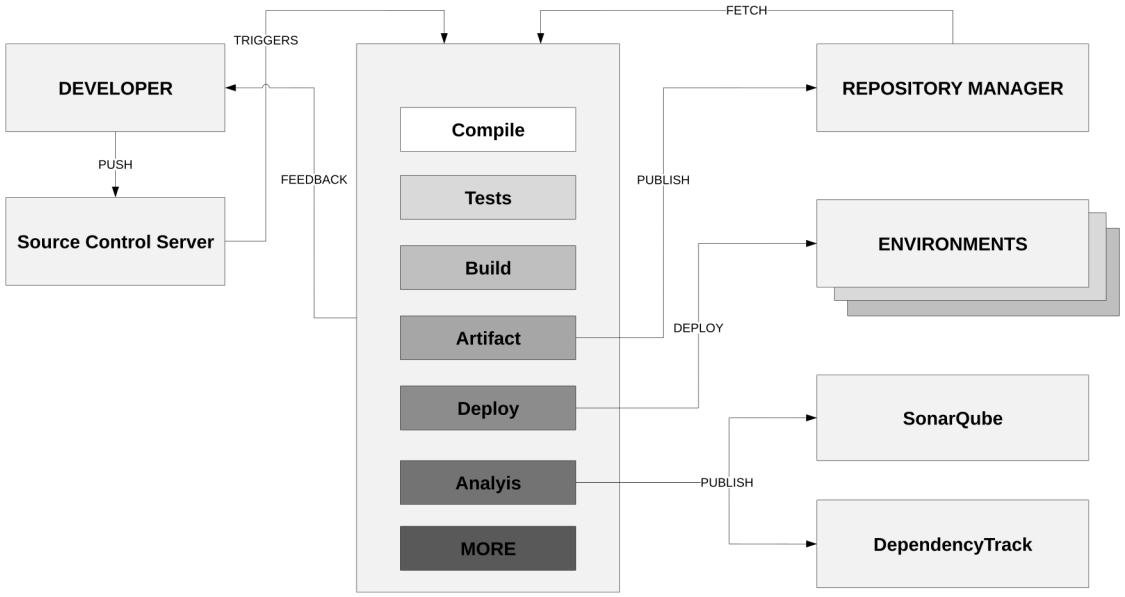
Continuous Delivery

- Continuous Delivery is an extension of the CI. It involves the Continuous Delivery of developed code to a staging environment as soon as the developer has developed it.
- It includes Continuous Integration, test automation, and deployment automation processes that enable fast and reliable software development and deployment with minimal manual effort.
- The core idea is to deliver the code to QA, customers, or any user base so that it can be continuously and regularly reviewed.
- This makes it possible to identify problems early in the development cycle and ensures that they can be resolved early.

Continuous Deployment

- Continuous Deployment is the next logical step after Continuous Delivery.
- It is the process of delivering code directly into production once it has been developed.
- With Continuous Deployment, all changes that pass the automated tests are automatically transferred to production.
- Successful implementation requires the automation of Continuous Integration, as well as the automation of Continuous Delivery into the staging environment. Finally, it also requires the ability to automatically deploy to production.

4.9.2 CI Workflow



Continuous Integration Workflow

In your CI workflow, you can integrate many useful tools depending on what your goal is. In this case, the following tools are part of the CI workflow.

Nexus[44]
: Manage binaries and build artifacts across your software supply chain.

SonarQube[45]
: Platform for static code analysis of the technical quality. Supports not only Java but also the analysis of other programming languages. Analyses the code for different quality ranges and displays the results graphically.

Renovate[46]
: Automated dependency updates for multi-platform and multi-languages.

DependencyTrack[47]
: Platform that allows organizations to identify and reduce risk from the use of third-party and open-source components.

Naikan[48]
: Naikan is a software inventory management tool driven by our CI/CD pipeline.

[44] https://www.sonatype.com/nexus
[45] http://www.sonarqube.org/
[46] https://github.com/renovatebot/renovate
[47] https://dependencytrack.org/
[48] https://naikan.io

4.9.3 Preconditions

Common code base

- To be able to integrate meaningfully within a team, a version management system must exist in which all developers can continuously integrate their changes.

Automated compiling

- Every integration must pass through uniformly defined tests before the changes are integrated.
- This requires automated compiling.

Continuous test development

- Each change should be developed with an associated test if possible.
- Newly developed tests should become part of the automated test suite.
- Use code coverage to document and control test coverage.

Frequent integration

- Integrate changes into the common code base as often as possible.
- The risk of failed integrations is minimized with small increments.
- Developer's work is often secured in the common code base.

Short test cycles

- Test cycle before integration should be short to encourage frequent integration.

Mirrored production environment

- The changes should be tested in a replica of the real production environment.

Easy access

- Even non-developers need easy access to the results of software development.

Automated reporting

- When was the last successful integration?
- What changes have been introduced since the last delivery?
- What is the quality of the version?

Automated distribution

4.9.4 Advantages and Disadvantages

Advantages:

- Integration problems are constantly being identified and resolved, not just before a milestone
- Early warnings for mismatched components
- Quick detection of errors through immediate execution of unit tests
- Constant availability of an executable stand for demo, test, or distribution purposes
- Fast feedback on the check-in of incorrect or incomplete code
- Early and frequent testing, errors are found earlier
- Tests take place in parallel with the development
- Correction of errors at the earliest, most favorable time
- Ability to react immediately to key metrics
- Review of *coding standards* and *best practices*

Disadvantages:

- Setting up the infrastructure
- Costs for hardware and software

4.9.5 Best Practices

- Use of a version management system
- Committing changes to the main branch should automatically trigger a build
- Check-in early and often, prefer Trunk Based Development[49]
- Do not check in a code that is not running
- Fix build errors immediately
- Tests should run as part of the CI process, with test failures causing a build failure
- Tackling problems early and failing quickly
- Code quality checks should run as part of the CI process
- Act and react based on metrics
- Create artifacts for each build
- Use configuration as code, the CI server configuration should be maintained as code
- Provide fast builds, so developers run these tests locally often

[49] https://trunkbaseddevelopment.com/

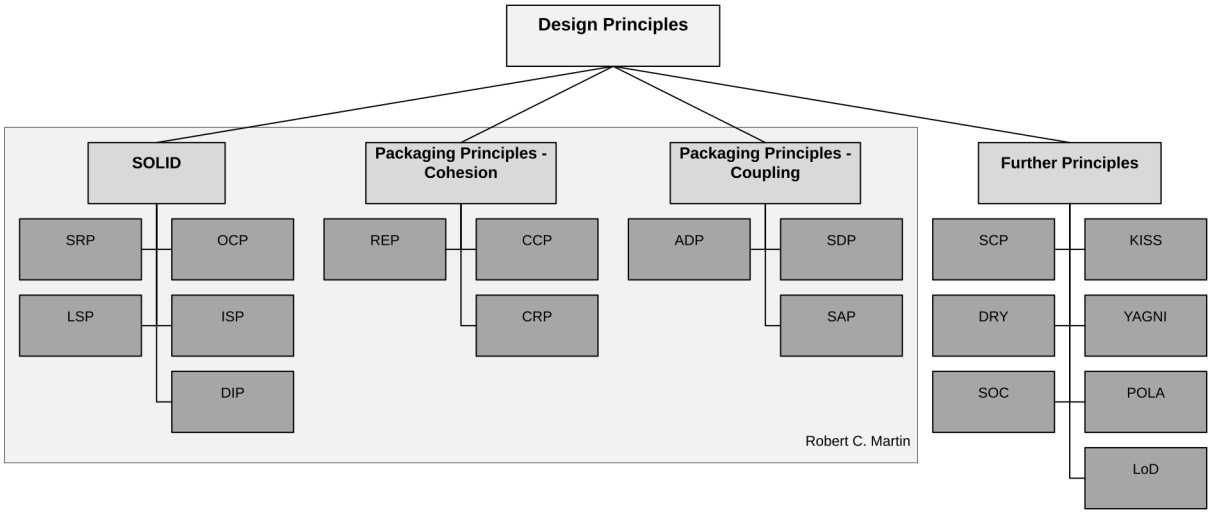
5. Design Principles

5.1 Goal of Design Principles

*Code must be **KISS**, **DRY** and **YAGNI**!*

- Design principles[1] should lead to good object-oriented design.
- Code and architecture smells are based on the violation of accepted design principles.
- Provide important information on how to fix code and architecture smells.
- If the violated design principle can be identified, it provides a first indication of what a better structure for the system might look like.
- Have been published and propagated by Robert C. Martin, Bertrand Meyer, and Barbara Liskov, among others.

5.2 Overview of Design Principles



Overview of Design Principles

SOLID Principles

- The acronym **SOLID** was created by Robert C. Martin for a group of these principles.
- According to Robert C. Martin, applying these principles leads to higher maintainability and thus to a longer service life of the software.
- Deal with the question of how to combine classes into modules and packages.
- Lead to high cohesion within modules and low coupling between modules.

[1] http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod

S	SRP	**S**ingle **R**esponsibility **P**rinciple	*A class should have one, and only one, reason to change.*
O	OCP	**O**pen **C**losed **P**rinciple	*You should be able to extend a classes behavior, without modifying it.*
L	LSP	**L**iskov **S**ubstitution **P**rinciple	*Derived classes must be substitutable for their base classes.*
I	ISP	**I**nterface **S**egregation **P**rinciple	*Make fine grained interfaces that are client specific.*
D	DIP	**D**ependency **I**nversion **P**rinciple	*Depend on abstractions, not on concretions.*

SOLID Design Principles

Packaging Principles - Cohesion

REP	**R**elease Reuse **E**quivalency **P**rinciple	*The granule of reuse is the granule of release.*
CCP	**C**ommon **C**losure **P**rinciple	*Classes that change together are packaged together.*
CRP	**C**ommon **R**euse **P**rinciple	*Classes that are used together are packaged together.*

Packaging Principles - Cohesion

Packaging Principles - Coupling

ADP	**A**cyclic **D**ependencies **P**rinciple	*The dependency graph of packages must have no cycles.*
SDP	**S**table **D**ependencies **P**rinciple	*Depend in the direction of stability.*
SAP	**S**table **A**bstractions **P**rinciples	*Abstractness increases with stability.*

Packaging Principles - Coupling

Further Design Principles

SCP	**S**peaking **C**ode **P**rinciple	*Writing code that speaks for itself and that does not need a comment.*
KISS	**K**eep **I**t **S**imple (and) **S**tupid	*Simplicity should be a key goal in design, and unnecessary complexity should be avoided.*
DRY	**D**on't **R**epeat **Y**ourself	*Every piece of knowledge must have a single, unambiguous, authoritative representation within a system.*
YAGNI	**Y**ou **A**in't **G**onna **N**eed **I**t!	*Always implement things when you actually need them, never when you just foresee that you need them.*
SOC	**S**eparation **O**f **C**oncerns	*Process of separating a program into distinct features that overlap in functionality as little as possible.*
POLA	**P**rinciple **O**f **L**east **A**stonishment	*If something has a hight astonishment factor, it may be necessary to redesign it.*
LoD	**L**aw **o**f **D**emeter	*Only talk to your immediate friends.*

Further Design Principles

5.3 SOLID Principles

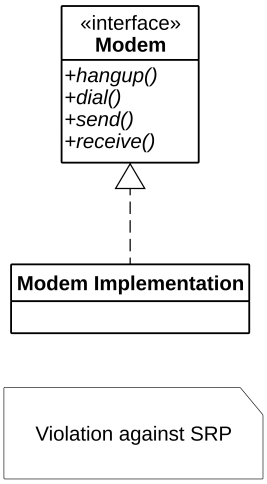
5.3.1 Single Responsibility Principle

A class should have one, and only one, reason to change.

- This is an object-oriented version of the functional programming principle that functions should have *no side effects*.
- A class should only have functions that directly contribute to the fulfillment of this task.
- One of the easiest ways to write classes that never need to be changed is to write classes that serve only one purpose.
- Each class should implement a coherent set of related functions.
- Having multiple responsibilities in a class leads to tight coupling and results in fragile designs.
- Following the principle of single responsibility is to ask yourself again and again whether each method and process of a class is directly related to the name of that class.
- For methods that do not match the name of the class, move these methods to other classes.

5.3.1.1 Example: `Modem`

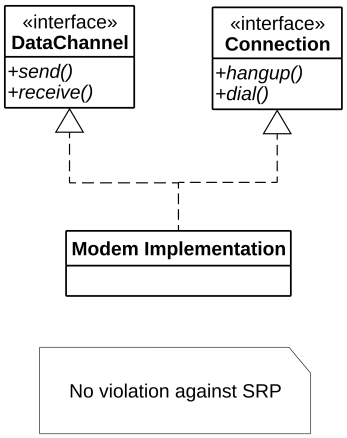
Noncompliant Code Example



Bad code

```
public interface Modem {
    // Connection management responsibilities
    void dial(String phoneNumber);
    void hangup();

    // Data management responsibilities
    void send(char c);
    char receive();
}
```

Compliant Solution



The Single Responsibility Principle (SRP) states that a class or module should have only one reason to change. In the given code example, the SRP violation is evident because the `Modem` interface is responsible for both connection management and data management.

To address this violation, the code can be refactored into a compliant solution that separates the responsibilities.

The `DataChannel` interface is created to handle the responsibilities related to data management. It declares the methods `send` and `receive` for sending and receiving data, respectively.

Good code

```
public interface DataChannel {
    void send(char c);
    char receive();
}
```

The `Connection` interface is created to handle the responsibilities related to connection management. It declares the methods `dial` and `hangup` for establishing and terminating a connection, respectively.

Good code

```
public interface Connection {
    void dial(String phoneNumber);
    void hangup();
}
```

The `Modem` class is modified to implement both the `Connection` and `DataChannel` interfaces. By implementing these interfaces, the `Modem` class takes on the responsibilities defined by each interface. It separates the connection management logic from the data management logic, adhering to the SRP.

By following this compliant solution, each interface and class has a single responsibility, making the code more maintainable and allowing for easier modification or extension of individual responsibilities without affecting the others.

Good code

```
public final class Modem implements Connection, DataChannel {

    @Override
    public void dial(String phoneNumber) {
         // logic
    }

    @Override
    public void hangup() {
         // logic
    }

    @Override
    public void send(char c) {
         // logic
    }

    @Override
    public char receive() {
         // logic
    }
}
```

5.3.1.2 Example: Book

Noncompliant Code Example

The `Book` class violates the SRP by having multiple responsibilities, such as storing book data and exporting the book to different formats.

Bad code

```
public record Book(String title, String author, String content) {

    public void exportToPdf() {
        // logic
    }

    public void exportToEpub() {
        // logic
    }
}
```

Compliant Solution

To address this violation, the code can be refactored into a compliant solution that separates the responsibilities. The `Book` class is modified to only handle the responsibility of storing book data. The export functionality is removed from the class.

Good code

```
public record Book(String title, String author, String content) {

}
```

Good code

```
public interface BookExporter {
    void export(Book book);
}
```

The `BookExporter` interface is introduced to handle the responsibility of exporting a book. It declares a single method `export` that takes a `Book` object as a parameter.

Good code

```
public final class PdfExporter implements BookExporter {

    @Override
    public void export(Book book) {
        // logic
    }
}
```

The `PdfExporter` class is created to handle the specific responsibility of exporting a book to PDF format. It implements the `BookExporter` interface and provides the implementation for the `export` method.

Good code

```
public final class EpubExporter implements BookExporter {

    @Override
    public void export(Book book) {
        // logic
    }
}
```

The `EpubExporter` class is created to handle the specific responsibility of exporting a book to EPUB format. It also implements the `BookExporter` interface and provides the implementation for the `export` method.

By following this compliant solution, the `Book` class has a single responsibility of storing book data, while the exporting functionality is moved to separate classes that implement the `BookExporter` interface.

5.3.1.3 Example: `Product`

Noncompliant Code Example

The `Product` class violates the SRP by having multiple responsibilities, which stores product data and also handles discount calculation.

Bad code

```
record Product(String name, double price) {

    public double calculateDiscount() {
        // Calculate the discount for the product.
        return 0.0;
    }
}
```

Compliant Solution

In the improved code, we separate the responsibilities of storing product data and discount calculation into two distinct classes. With this separation, the `Product` class focuses solely on storing product data, and the `DiscountCalculator` class is responsible for discount calculation. This adheres to the SRP, making the code more modular and maintainable.

Good code

```
record Product(String name, double price) {

}
```

Good code

```
class DiscountCalculator {
    public double calculateDiscount(Product product) {
        // Calculate the discount for the product.
        return 0.0;
    }
}
```

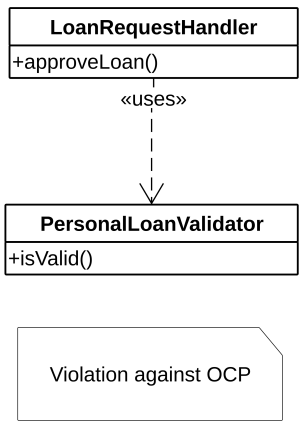
5.3.2 Open Closed Principle

You should be able to extend a classes behavior, without modifying it.

- Classes that follow the OCP have two essential characteristics:
  - **Open to extensions**, the behavior of such class can be extended to meet new requirements of an existing or even new application.
  - **Closed against subsequent modification**, no longer need to be modified to serve as a basis for new requirements.
- Functions that perform different actions due to type switches are good examples of OCP violations. Such functions are never closed against changes because adding a new type requires changing the source code of the function.
- It is more elegant and robust to extend a class group by adding a class than to modify the existing source code.
- By using e.g. abstract base classes, software components can be created which have a fixed unchangeable implementation, but whose behavior can be changed indefinitely by inheritance and polymorphism.

5.3.2.1 Example: `LoanRequestHandler`

Noncompliant Code Example



The noncompliant code violates the OCP because the `LoanRequestHandler` class is not closed for modification.

Bad code

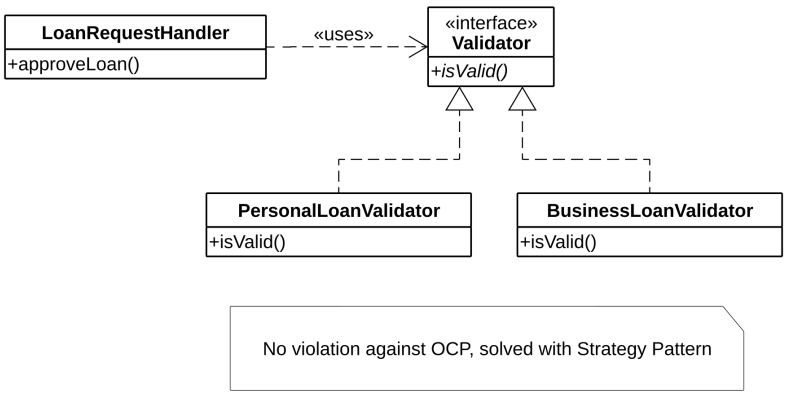
```
public class LoanRequestHandler {
    private final int balance;
    private final int period;

    public LoanRequestHandler(int balance, int period) {
        this.balance = balance;
        this.period = period;
    }

    public void approveLoan(PersonalLoanValidator validator) {
        if(validator.isValid(balance)) {
            System.out.println("Loan approved...");
        } else {
            System.out.println("Sorry not enough balance...");
        }
    }
}
```

Bad code

```
public final class PersonalLoanValidator {
    public boolean isValid(int balance) {
        return balance > 1000;
    }
}
```

The `LoanRequestHandler` class has a method `approveLoan` that takes a `PersonalLoanValidator` object as a parameter. This tightly couples the class to a specific type of validator, violating the OCP. If we need to introduce new types of loan validators, we would have to modify the `LoanRequestHandler` class, which is against the OCP principle.

Compliant Solution



Good code

```
public class LoanRequestHandler {
    private final int balance;
    private final int period;

    public LoanRequestHandler(int balance, int period) {
        this.balance = balance;
        this.period = period;
    }

    public void approveLoan(Validator validator) {
        if (validator.isValid(balance)) {
            System.out.println("Loan approved...");
        } else {
            System.out.println("Sorry not enough balance...");
        }
    }
}
```

The `approveLoan` method of the `LoanRequestHandler` class is modified to accept a `Validator` interface instead of a specific implementation. This change allows the class to work with any object that implements the `Validator` interface, making it open for extension.

Good code

```
public interface Validator {
    boolean isValid(int balance);
}
```

The `Validator` interface is introduced to define the contract for loan validators. It declares a single method `isValid` that takes the balance as a parameter and returns a boolean value indicating whether the loan is valid.

Good code

```
public class PersonalLoanValidator implements Validator {
    public boolean isValid(int balance) {
        return balance > 1000;
    }
}
```

The `PersonalLoanValidator` class is created as an implementation of the `Validator` interface. It provides the specific implementation of the `isValid` method for personal loans.

Good code

```
public class BusinessLoanValidator implements Validator {
    public boolean isValid(int balance) {
        return balance > 5000;
    }
}
```

The `BusinessLoanValidator` class is created as another implementation of the `Validator` interface. It provides a different implementation of the `isValid` method for business loans.

By following this compliant solution, the `LoanRequestHandler` class remains closed for modification as new validators can be introduced by implementing the `Validator` interface. The class can work with any object that adheres to the `Validator` interface, making it open for extension

5.3.2.2 Example: `Shape`

Noncompliant Code Example

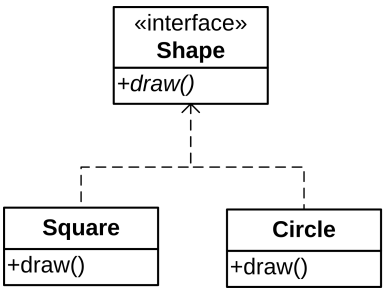
The noncompliant code violates the OCP by using a switch statement to handle different types of shapes, which requires modifying the `draw` method whenever a new shape is introduced.

Bad code

```
public void draw(Shape[] shapes) {
    for (Shape shape : shapes) {
        switch (shape.type()) {
            case Shape.SQUARE -> draw((Square) shape);
            case Shape.CIRCLE -> draw((Circle) shape);
        }
    }
}

private void draw(Circle circle) {
    // logic for circle
}

private void draw(Square square) {
    // logic for square
}
```

The `draw` method takes an array of `Shape` objects and uses a switch statement to determine the shape type and invoke the corresponding `draw` method. This approach violates the OCP because whenever a new shape type is added, the `draw` method needs to be modified to include a new case in the switch statement, which results in the method being open for modification.

Compliant Solution



Good code

```
public interface Shape {
    void draw();
}
```

The `Shape` interface is introduced to define the contract for shapes. It declares a single method `draw` that is responsible for drawing the shape.

Good code

```
public class Square implements Shape {
    @Override
    public void draw() {}
}
```

Good code

```
public class Circle implements Shape {
    @Override
    public void draw() {}
}
```

Good code

```
public void draw(Shape[] shapes) {
    for (Shape shape : shapes) {
        shape.draw();
    }
}
```

The `draw` method is modified to iterate over the array of `Shape` objects and invoke the draw method on each shape. By doing so, the method is closed for modification as it doesn't need to be updated whenever a new shape is introduced. The implementation of the `draw` method for each shape is determined by the specific shape class, which follows the OCP by allowing for extension without modification.

5.3.2.3 Example: `HumanResourceDepartment`

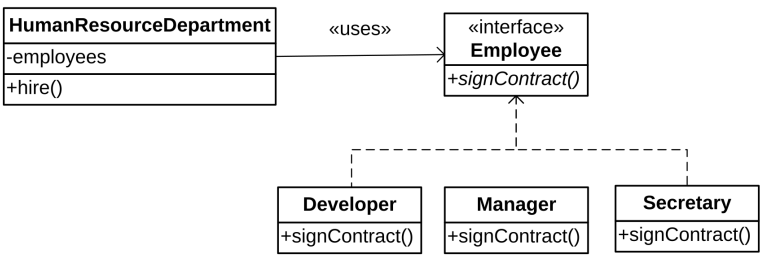
Noncompliant Code Example

Bad code

```
public class HumanResourceDepartment {

    private final List<Developer> developers;
    private final List<Manager> managers;

    public HumanResourceDepartment() {
        this.developers = new ArrayList<>();
        this.managers = new ArrayList<>();
    }

    public void hire(Developer developer) {
        developer.signContract();
        this.developers.add(developer);
    }

    public void hire(Manager manager) {
        manager.signContract();
        this.managers.add(manager);
    }
}
```

The `HumanResourceDepartment` class has specific methods for hiring developers and managers. This tightly couples the class to specific employee types, violating the OCP. If we need to introduce new types of employees (e.g., `Secretary`), we would have to modify the `HumanResourceDepartment` class, which is against the OCP principle.

Compliant Solution



Good code

```
interface Employee {
    void signContract();
}
```

The `Employee` interface is introduced to define the contract for employees. It declares a single method `signContract` that is responsible for signing the contract.

Good code

```
class Developer implements Employee {
    public void signContract() {}
}
```

Good code

```
class Manager implements Employee {
    public void signContract() {}
}
```

Good code

```
class Secretary implements Employee {
    public void signContract() {}
}
```

Good code

```
public class HumanResourceDepartment {
    private final List<Employee> employees;

    public HumanResourceDepartment() {
        this.employees = new ArrayList<>();
    }

    public void hire(Employee employee) {
        employee.signContract();
        this.employees.add(employee);
    }
}
```

The `HumanResourceDepartment` class is modified to work with the `Employee` interface instead of specific employee types. The `hire` method now takes an Employee object as a parameter and invokes the `signContract` method on the employee. This change allows the class to work with any object that implements the `Employee` interface, making it open for extension.

5.3.2.4 Example: `Calculator`

Noncompliant Code Example

Bad code

```
public class Calculator {
    public double calculate(double x, double y, String operator) {
        double result;

        if ("plus".equals(operator)) {
            result = x + y;
        } else if ("minus".equals(operator)) {
            result = x - y;
        } else if ("multiply".equals(operator)) {
            result = x * y;
        } else if ("divide".equals(operator)) {
            result = x / y;
        } else {
            throw new IllegalArgumentException(String.format("Operator %s not supported!", operator));
        }

        return result;
    }
}
```

The `Calculator` class has a `calculate` method that performs different operations based on the value of the operator parameter. However, if we need to introduce new operations (e.g., exponentiation, square root), we would have to modify the `Calculator` class, which is against the OCP principle.

Compliant Solution

Good code

```
public enum Operation {
    PLUS((x, y) -> x + y),
    MINUS((x, y) -> x - y),
    MULTIPLY((x, y) -> x * y),
    DIVIDE((x, y) -> x / y);

    private final DoubleBinaryOperator operator;

    Operation(DoubleBinaryOperator operator) {
        this.operator = operator;
    }

    public double apply(double x, double y) {
        return this.operator.applyAsDouble(x, y);
    }
}
```

An `enum` called `Operation` is introduced to represent different operations. Each operation is defined as a constant with a corresponding lambda expression (`DoubleBinaryOperator`) that performs the operation. The apply method is provided to `apply` the operation to the given operands.

By using an `enum` and lambdas, new operations can be easily added by defining additional constants in the `Operation` enum without modifying the existing code. This adheres to the OCP principle, as the `Calculator` class remains closed for modification but open for extension.

Good code

```
public class Calculator {
    public double calculate(double x, double y, Operation operation) {
        return operation.apply(x, y);
    }
}
```

The `calculate` method now takes an `Operation` object as a parameter and directly applies the operation to the given operands. This change allows the class to work with any operation defined in the `Operation` enum, making it open for extension.

5.3.2.5 Example: `FileParser`

Noncompliant Code Example

Bad code

```
public class FileParser {

    public String parse(String filePath) {
        if (filePath.indexOf(".xml") > 1) {
            return parseXML(filePath);
        } else if (filePath.indexOf(".json") > 1) {
            return parseJson(filePath);
        }

        return null;
    }

    private String parseXML(String filePath) {
        // logic
        return null;
    }

    private String parseJson(String filePath) {
        // logic
        return null;
    }
}
```

The `FileParser` class has a `parse` method that determines the file type based on its extension and then calls the respective private parse methods (`parseXML` and `parseJson`). This approach violates the OCP because whenever a new file type needs to be supported, the `FileParser` class needs to be modified to add a new condition and corresponding parsing logic.

Compliant Solution

Good code

```
public interface FileParser {
    String parse(String file);
    boolean supports(String extension);
}
```

An interface `FileParser` is introduced, which defines two methods:`parse` for parsing the file content and `supports` for determining whether the parser supports a given file extension.

Good code

```
public class JsonParser implements FileParser {

    @Override
    public String parse(String file) {
        return String.format("Parser %s", JsonParser.class.getSimpleName());
    }
    @Override
    public boolean supports(String extension) {
        return "json".equalsIgnoreCase(extension);
    }
}
```

Good code

```
public class XmlParser implements FileParser {

    @Override
    public String parse(String file) {
        return String.format("Parser %s", XmlParser.class.getSimpleName());
    }
    @Override
    public boolean supports(String extension) {
        return "xml".equalsIgnoreCase(extension);
    }
}
```

Good code

```
public final class FileParserFactory {

    private FileParserFactory() {
    }

    public static FileParser newInstance(String extension) {
        ServiceLoader<FileParser> loader = ServiceLoader.load(FileParser.class);

        for (FileParser parser : loader) {
            if (parser.supports(extension)) {
                return parser;
            }
        }
        return null;
    }
}
```

The `FileParserFactory` class is responsible for creating instances of `FileParser` based on the provided file extension. It uses the `ServiceLoader` mechanism to dynamically load implementations of the `FileParser` interface.

Good code

```
// META-INF/services/swcs.dp.ocp.parser.after.FileParser
swcs.dp.ocp.parser.after.JsonParser
swcs.dp.ocp.parser.after.XmlParser
```

For the Service API mechanism, a file named `META-INF/services/swcs.dp.ocp.parser.after.FileParser` is created, which lists the fully qualified names of the available `FileParser` implementations.

Good code

```
class Client {

    public static void main(String[] args) {
        FileParser parser = FileParserFactory.newInstance("xml");
        System.out.println(parser.parse("test.xml"));
    }
}
```

5.3.3 Liskov Substitution Principle

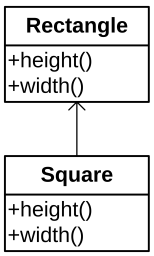
Derived classes must be substitutable for their base classes.

Let q(x) be a property provable about objects x of type T. Then q(y) should be true for objects y of type S, where S is a subtype of T.

- In a clean OOD, the derivation relationship between superclass and subclass represents a so-called **is-a** relationship, i.e. an object of the subclass is compatible in type with a superclass object and can be used wherever a superclass object is required.
- This ensures that superclass type operations applied to a subclass object are performed correctly.
- In this case, it is always safe to replace a superclass-type object with a subclass-type object.
- When inheriting, do not only pay attention to the data but also consider the behavior of the methods of a class.

5.3.3.1 Example: `Rectangle`

Noncompliant Code Example



Bad code

```
1  public class Rectangle {
2
3      private double height;
4      private double width;
5
6      public double height() {
7          return this.height;
8      }
9
10     public double width() {
11         return this.width;
12     }
13
14     public void height(double height) {
15         this.height = height;
16     }
17
18     public void width(double width) {
19         this.width = width;
20     }
21
```

```
22      public double area() {
23          return this.width * this.height;
24      }
25  }
```

The `Rectangle` class has separate `height` and `width` properties, and its area method calculates the `area` based on those properties. The `Square` class extends `Rectangle` and overrides the `area` method to calculate the area based on the height only (as all sides of a square are equal). However, this violates the LSP because a square is not a proper subtype of a rectangle in terms of behavior.

Bad code

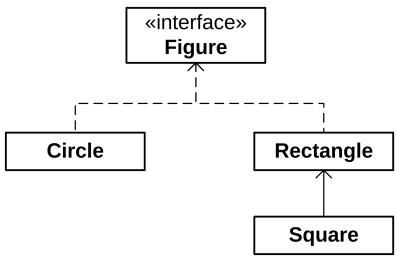
```
1   public class Square extends Rectangle {
2
3       @Override
4       public double area() {
5           return this.height() * this.height();
6       }
7   }
```

Bad code

```
1   public void do(Rectangle rectangle) {
2       rectangle.width(5);
3       rectangle.height(4);
4
5       if (rectangle.area() != 20) {
6           throw new IllegalStateException("Error in area calculation!");
7       }
8   }
```

The `do` method accepts a `Rectangle` object as a parameter and performs some operation. Since both `Rectangle` and `Square` are subclasses of `Figure`, the `do` method can not work with either a rectangle or a square object without affecting the correctness of the program.

Compliant Solution



Good code

```
public interface Figure {
    double area();
}
```

Good code

```
public class Circle implements Figure {
    private final double radius;

    public Circle(double radius) {
        this.radius = radius;
    }
    @Override
    public double area() {
        return Math.PI * (this.radius * this.radius);
    }
}
```

Good code

```
public class Rectangle implements Figure {
    private final double length;
    private final double width;

    public Rectangle(double length, double width) {
        this.length = length;
        this.width = width;
    }
    @Override
    public double area() {
        return this.length * this.width;
    }
}
```

Good code

```
public class Square extends Rectangle {
    public Square(double side) {
        super(side, side);
    }
}
```

The `Square` class is modified to extend the `Rectangle` class. Since a square is a special case of a rectangle where all sides are equal, the `Square` class uses the `Rectangle` class by providing the same value for both length and width in its constructor.

Good code

```
public void do(Rectangle rectangle) {
    // Rectangle double and width was set through constructor

    if (rectangle.area() != 20) {
        throw new IllegalStateException("Error in area calculation!");
    }
}
```

5.3.3.2 Example: `Coupon`

Noncompliant Code Example

The noncompliant code violates the LSP because the `PromotionCoupon` class modifies the behavior of the `afterDiscount` method, strengthening its pre-conditions.

Bad code

```
public class Coupon {

    private final BigDecimal regularPrice;

    public Coupon(BigDecimal regularPrice) {
        this.regularPrice = regularPrice;
    }

    public BigDecimal regularPrice() {
        return this.regularPrice;
    }

    public BigDecimal afterDiscount(BigDecimal discount) {
        if (discount == null) {
            return this.regularPrice;
        }

        if (this.regularPrice.compareTo(discount) > 0) {
            return this.regularPrice.subtract(discount);
        }

        return BigDecimal.ZERO;
    }
}
```

Bad code

```
public class PromotionCoupon extends Coupon {

    private static final double PERCENTAGE_DISCOUNT = 0.9;

    public PromotionCoupon(BigDecimal regularPrice) {
        super(regularPrice);
    }

    @Override
    public BigDecimal afterDiscount(BigDecimal discount) {
        // strengthened pre-conditions and no check for null
        if (regularPrice().compareTo(discount) < 0) {
            throw new IllegalArgumentException("Discount can not be greater than the price!");
        }

        return super.afterDiscount(discount)
                .multiply(BigDecimal.valueOf(PERCENTAGE_DISCOUNT));
    }
}
```

The `Coupon` class represents a regular coupon with a `regularPrice` and an `afterDiscount` method that calculates the price after applying a discount. The `PromotionCoupon` class extends Coupon and overrides the `afterDiscount` method to include an additional check for the discount, throwing an exception if the discount is greater than the regular price.

The noncompliant code violates the LSP because the behavior of the `afterDiscount` method is modified in the `PromotionCoupon` subclass, strengthening its pre-conditions. As a result, when the `PromotionCoupon` class is used in place of the `Coupon` class, it throws an exception (`IllegalArgumentException`) in cases where the base class implementation would return a valid value.

Bad code

```
class Client {

    private static final BigDecimal REGULAR_PRICE = BigDecimal.valueOf(8.99);

    public static void main(String[] args) {
        testCoupon(new Coupon(REGULAR_PRICE), null);                          // Pay 8,990000
        testCoupon(new PromotionCoupon(REGULAR_PRICE), null);                 // NPE

        testCoupon(new Coupon(REGULAR_PRICE), BigDecimal.valueOf(5));         // Pay 3,990000
        testCoupon(new PromotionCoupon(REGULAR_PRICE), BigDecimal.valueOf(5)); // Pay 3,591000

        testCoupon(new Coupon(REGULAR_PRICE), BigDecimal.valueOf(10));         // You get it for free!
        testCoupon(new PromotionCoupon(REGULAR_PRICE), BigDecimal.valueOf(10)); // IAE
    }

    private static void testCoupon(Coupon coupon, BigDecimal discount) {
        BigDecimal price = coupon.afterDiscount(discount);

        // Client code relies on Coupon implementation
        if (BigDecimal.ZERO.compareTo(price) == 0) {
            System.out.println("You get it for free!");
        } else {
            System.out.printf("Pay %f%n", price);
        }
    }
}
```

Compliant Solution

In the compliant solution, the LSP violation is resolved by introducing the `Coupon` interface and providing a clear contract for the behavior of the `afterDiscount` method:

Good code

```
public interface Coupon {

    BigDecimal regularPrice();

    /**
     * Returns a {@code BigDecimal} whose value is {@code (+this)},
     * minus the provided discount.
     *
     * @param discount the discount to use.
     * @return {@code this}, minus the discount. If the discount is
```

```
     *         greater than the regular price {@code BigDecimal.ZERO} will be returned.
     *         If the discount is {@code null} than this discount will not be subtracted.
     */
    BigDecimal afterDiscount(BigDecimal discount);
}
```

The `RegularCoupon` class implements the `Coupon` interface and provides the basic behavior of a regular coupon:

Good code

```
public class RegularCoupon implements Coupon {

    private final BigDecimal regularPrice;

    public RegularCoupon(BigDecimal regularPrice) {
        this.regularPrice = regularPrice;
    }

    @Override
    public BigDecimal regularPrice() {
        return this.regularPrice;
    }

    @Override
    public BigDecimal afterDiscount(BigDecimal discount) {
        if (discount == null) {
            return this.regularPrice;
        }

        if (this.regularPrice.compareTo(discount) > 0) {
            return this.regularPrice.subtract(discount);
        }

        return BigDecimal.ZERO;
    }
}
```

The `PromotionCoupon` class extends `RegularCoupon` and inherits its behavior without modifying the pre-conditions. It only overrides the method to apply an additional discount percentage:

Good code

```
public final class PromotionCoupon extends RegularCoupon {

    private static final double PERCENTAGE_DISCOUNT = 0.9;

    public PromotionCoupon(BigDecimal regularPrice) {
        super(regularPrice);
    }

    @Override
    public BigDecimal afterDiscount(BigDecimal discount) {
        return super.afterDiscount(discount)
                .multiply(BigDecimal.valueOf(PERCENTAGE_DISCOUNT));
    }
}
```

In the compliant solution, the client code can work with any object that implements the `Coupon` interface, including both `RegularCoupon` and `PromotionCoupon`. This ensures that the client code is not affected by the specific implementation details of the subclasses and allows for substitutability without violating the LSP.

Good code

```
class Client {

    private static final BigDecimal REGULAR_PRICE = BigDecimal.valueOf(8.99);

    public static void main(String[] args) {
        testCoupon(new RegularCoupon(REGULAR_PRICE), null);                    // Pay 8,990000
        testCoupon(new PromotionCoupon(REGULAR_PRICE), null);                  // Pay 8,091000

        testCoupon(new RegularCoupon(REGULAR_PRICE), BigDecimal.valueOf(5));   // Pay 3,990000
        testCoupon(new PromotionCoupon(REGULAR_PRICE), BigDecimal.valueOf(5)); // Pay 3,591000

        testCoupon(new RegularCoupon(REGULAR_PRICE), BigDecimal.valueOf(10));  // You get it for free!
        testCoupon(new PromotionCoupon(REGULAR_PRICE), BigDecimal.valueOf(10)); // You get it for free!
    }

    private static void testCoupon(Coupon coupon, BigDecimal discount) {
        BigDecimal price = coupon.afterDiscount(discount);

        if (BigDecimal.ZERO.compareTo(price) == 0) { // Client code relies on Coupon implementation
            System.out.println("You get it for free!");
        } else {
            System.out.printf("Pay %f%n", price);
        }
    }
}
```

5.3.3.3 Example: `Bird`

Noncompliant Code Example

The noncompliant code violates the LSP because the `Penguin` class overrides the `fly()` method to throw an `UnsupportedOperationException` because penguins cannot fly. When we call the `fly()` method on a `Penguin` object, it throws an exception, indicating that penguins cannot fly.

Bad code

```
class Bird {
    private final String name;

    public Bird(String name) {
        this.name = name;
    }

    public String name() {
        return this.name;
    }

    public void eat() {
        System.out.println(name + " is eating.");
    }

    public void fly() {
        System.out.println(name + " is flying.");
    }
}
```

Bad code

```
class Sparrow extends Bird {
    public Sparrow(String name) {
        super(name);
    }

    @Override
    public void eat() {
        System.out.println(name() + " is pecking at seeds.");
    }
}
```

Bad code

```
class Penguin extends Bird {
    public Penguin(String name) {
        super(name);
    }

    @Override
    public void fly() {
        throw new UnsupportedOperationException(name() + " can't fly.");
    }
}
```

Compliant Solution

In the compliant solution, the LSP violation is resolved, both Sparrow and Penguin classes override the `move()` method instead of the `fly()` method. This allows them to represent their specific modes of movement, where the `Sparrow` flies, and the `Penguin` swims. The `Penguin` class no longer throws an exception because it provides a meaningful implementation for the `move()` method.

By making this change, you align the class hierarchy with the real-world characteristics of the birds, and the Liskov Substitution Principle is maintained because you can use objects of both derived classes interchangeably without causing issues in the program.

Good code

```
class Bird {
    private final String name;

    public Bird(String name) {
        this.name = name;
    }

    public String name() {
        return this.name;
    }

    public void eat() {
        System.out.println(name + " is eating.");
    }

    public void move() {
        System.out.println(name + " is moving.");
    }
}
```

Good code

```
class Sparrow extends Bird {
    public Sparrow(String name) {
        super(name);
    }

    @Override
    public void move() {
        System.out.println(name() + " is flying.");
    }
}
```

Good code

```
class Penguin extends Bird {
    public Penguin(String name) {
        super(name);
    }

    @Override
    public void move() {
        System.out.println(name() + " is swimming gracefully.");
    }
}
```

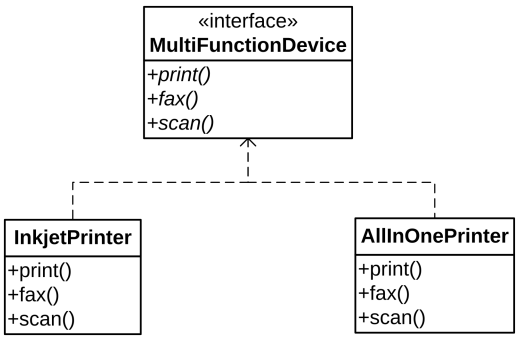
5.3.4 Interface Segregation Principle

Make fine-grained interfaces that are client-specific.

- Avoidance of *interface pollution* and *fat* interfaces.
- Instead of interfaces with all possible methods that clients might need, there should be separate interfaces that cover the specific needs of each client type.
- Methods of individual interfaces should have low coupling.
- Clients should not depend on methods that are not needed at all.
- Distribution should be made according to the requirements of the clients for the interfaces.
- Clients must therefore only operate with interfaces that can only do what they need.
- Enables software to be divided into decoupled and therefore flexible classes.
- Future professional or technical requirements therefore only require minor changes to the software itself.

5.3.4.1 Example: `MultiFunctionDevice`

Noncompliant Code Example



The noncompliant code example violates the Interface Segregation Principle (ISP). It defines a single interface `MultiFunctionDevice` that includes methods for printing, faxing, and scanning. This design forces classes that implement this interface to implement all the methods, even if they don't support or need all the functionalities. This leads to unnecessary dependencies and potential issues if a client tries to call unsupported methods.

Bad code

```
public interface MultiFunctionDevice {
    void print();
    void fax();
    void scan();
}
```

The `AllInOnePrinter` class implements this interface and provides implementations for all the methods.

Bad code

```
class AllInOnePrinter implements MultiFunctionDevice {
    @Override
    public void print() {
    }

    @Override
    public void fax() {
    }

    @Override
    public void scan() {
    }
}
```

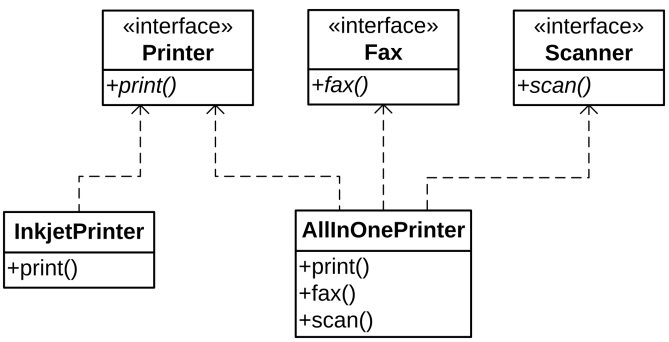
The `InkjetPrinter` class also implements the `MultiFunctionDevice` interface but throws an `UnsupportedOperationException` for the `fax()` and `scan()` methods since it doesn't support those functionalities.

Bad code

```
class InkjetPrinter implements MultiFunctionDevice {

    @Override
    public void print() {
    }

    @Override
    public void fax() {
        throw new UnsupportedOperationException();
    }

    @Override
    public void scan() {
        throw new UnsupportedOperationException();
    }
}
```

This design violates the ISP because it forces all classes implementing `MultiFunctionDevice` to provide implementations for all three methods, even if some methods are not applicable to them.

Compliant Solution



The compliant solution resolves the violation by defining separate interfaces for each functionality: `Printer`, `Fax`, and `Scanner`.

Good code

```
public interface Printer {
    void print();
}
```

Good code

```
public interface Fax {
    void fax();
}
```

Good code

```
public interface Scanner {
    void scan();
}
```

The `InkjetPrinter` class now only implements the `Printer` interface, as it doesn't support faxing or scanning.

Good code

```
class InkjetPrinter implements Printer {

    @Override
    public void print() {
    }
}
```

The `AllInOnePrinter` class implements all three interfaces, `Printer`, `Fax`, and `Scanner`, as it supports all the functionalities.

Good code

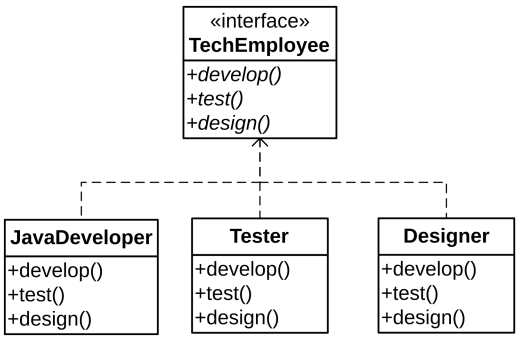
```
class AllInOnePrinter implements Printer, Fax, Scanner {

    @Override
    public void print() {
    }

    @Override
    public void fax() {
    }

    @Override
    public void scan() {
    }
}
```

By splitting the functionalities into separate interfaces, each class can implement only the interfaces relevant to their capabilities. This adheres to the ISP, as clients can depend on the specific interfaces they require, reducing unnecessary dependencies and potential issues with unsupported functionalities.

5.3.4.2 Example: `TechEmployee`

Noncompliant Code Example

The noncompliant code example violates the Interface Segregation Principle (ISP). It defines a single interface `TechEmployee` that includes methods for developing, testing, and designing. This design forces classes that implement this interface to implement all the methods, even if they don't support or need all the functionalities. This leads to unnecessary dependencies and potential issues if a client tries to call unsupported methods.



Bad code

```
public interface TechEmployee {
    void develop();
    void test();
    void design();
}
```

The `JavaDeveloper` class implements this interface and provides an implementation for the `develop()` method but throws an `UnsupportedOperationException` for the `test()` and `design()` methods since it doesn't support those functionalities.

Bad code

```
public class JavaDeveloper implements TechEmployee {

    @Override
    public void develop() {
        System.out.println("Yes, I love it!");
    }

    @Override
    public void test() {
        throw new UnsupportedOperationException();
    }

    @Override
    public void design() {
        throw new UnsupportedOperationException();
    }
}
```

The `Tester` class implements the `TechEmployee` interface and provides an implementation for the `test()` method but throws an `UnsupportedOperationException` for the `develop()` and `design()` methods.

Bad code

```
public class Tester implements TechEmployee {

    @Override
    public void develop() {
        throw new UnsupportedOperationException();
    }

    @Override
    public void test() {
        System.out.println("Yes, I love it!");
    }

    @Override
    public void design() {
        throw new UnsupportedOperationException();
    }
}
```

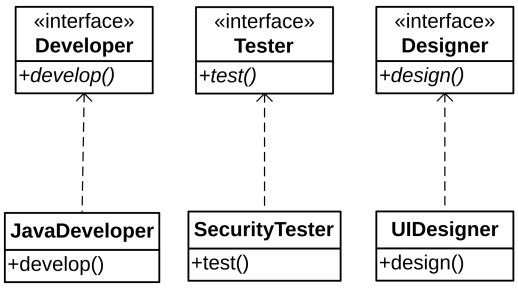
The `Designer` class implements the `TechEmployee` interface and provides an implementation for the `design()` method but throws an `UnsupportedOperationException` for the `develop()` and `test()` methods.

Bad code

```
public class Designer implements TechEmployee {

    @Override
    public void develop() {
        throw new UnsupportedOperationException();
    }

    @Override
    public void test() {
        throw new UnsupportedOperationException();
    }

    @Override
    public void design() {
        System.out.println("Yes, I love it!");
    }
}
```

This design violates the ISP because it forces all classes implementing `TechEmployee` to provide implementations for all three methods, even if some methods are not applicable to them.

Compliant Solution



The compliant solution resolves the violation by defining separate interfaces for each functionality: `Developer`, Tester, `and` Designer‘.

Good code

```
public interface Developer {
    void develop();
}
```

Good code

```
public interface Designer {
    void design();
}
```

Good code

```
public interface Tester {
    void test();
}
```

The `JavaDeveloper` class now only implements the `Developer` interface, as it only supports development.

Good code

```
public class JavaDeveloper implements Developer {

    @Override
    public void develop() {
        System.out.println("Yes, I love it!");
    }
}
```

The `SecurityTester` class now only implements the `Tester` interface, as it only supports testing.

Good code

```
public class SecurityTester implements Tester {

    @Override
    public void test() {
        System.out.println("Yes, I love it!");
    }
}
```

The `UIDesigner` class now only implements the `Designer` interface, as it only supports designing.

Good code

```
public class UIDesigner implements Designer {

    @Override
    public void design() {
        System.out.println("Yes, I love it!");
    }
}
```

By splitting the functionalities into separate interfaces, each class can implement only the interfaces relevant to their capabilities.

5.3.4.3 Example: `StockOrder`

Noncompliant Code Example

The noncompliant code example violates the Interface Segregation Principle (ISP) by defining a single interface `StockOrder` that includes multiple methods with different combinations of parameters. This design forces the clients to implement and provide unnecessary parameter combinations, leading to a bloated interface and potential confusion.

Bad code

```
public interface StockOrder {

    void sell(StockId id, double price, int quantity);

    void buy(StockId id, int quantity, double price);

    void buy(StockId id, int quantity, double price, double commission);

    void buy(StockId id, int quantity, double minPrice, double maxPrice, double commission);

}
```

Compliant Solution

The compliant solution resolves the violation by redefining the `StockOrder` interface with two methods: `sell()` and `buy()`, both using a `Price` object to represent the price parameter.

Good code

```
public interface StockOrder {

    void sell(StockId id, int quantity, Price price);

    void buy(StockId id, int quantity, Price price);

}
```

The `Price` object encapsulates the price value, ensuring a consistent representation and avoiding the need for multiple method signatures with different parameter combinations.

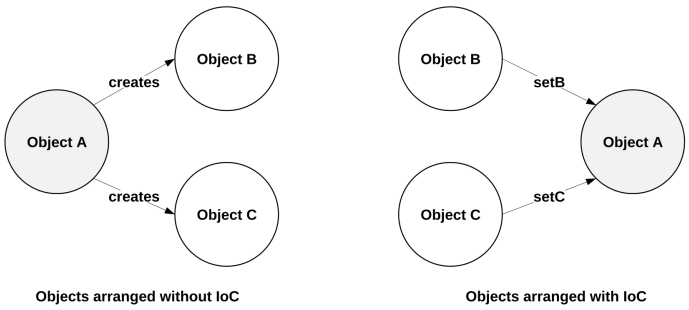
By using the `Price` object, both the `sell()` and `buy()` methods can accept the same type of parameter for price, simplifying the interface and avoiding the need for overloaded methods.

Clients implementing the `StockOrder` interface can now provide the necessary parameters without being burdened with unnecessary combinations, improving code clarity and reducing the chances of errors.

5.3.5 Dependency Inversion Principle

Depend on abstractions, not on concretions.

- Dependencies should always be directed from more concrete modules of **lower** levels to abstract modules of **higher** levels.
- No class should instantiate foreign classes but should receive them as abstractions (e.g. interfaces) in the form of a parameter.
- This ensures that the dependency relationships always run in one direction, from the concrete to the abstract modules, from the derived classes to the base classes.
- **Hollywood principle**: *Don't call us, we'll call you!*
- Dependencies between objects are controlled from the outside.
  - Reduction of dependencies between the modules
  - Avoidance of cyclical dependencies
  - Decoupling of the component from their environment
  - Reduction of the necessary knowledge



5.3.5.1 Example: `UserService`

Noncompliant Code Example

The noncompliant code example violates the Dependency Inversion Principle (DIP) by directly creating an instance of `UserRepository` within the `UserService` class. This creates a tight coupling between the `UserService` and the concrete implementation of `UserRepository`, making it difficult to change or extend the behavior of `UserService` in the future.

Bad code

```
public class UserService {
    private final UserRepository userRepository;

    public UserService() {
        this.userRepository = new UserRepositoryHibernate();
    }
}
```

This design violates the DIP because it directly depends on the concrete implementation `UserRepositoryHibernate`. This tight coupling makes it challenging to substitute or change the implementation of `UserRepository` without modifying the `UserService` class.

Compliant Solution

In the compliant solution, the `UserService` class no longer creates an instance of `UserRepository` directly. Instead, the `UserRepository` is passed as a constructor parameter, allowing the dependency to be injected from the outside.

By using constructor injection, the `UserService` class becomes decoupled from the specific implementation of `UserRepository`. This enables flexibility and modularity, as different implementations of `UserRepository` can be easily provided to the `UserService` class without requiring any modifications to its code.

Good code

```
public class UserService {
    private final UserRepository userRepository;

    public UserService(UserRepository userRepository) {
         this.userRepository = userRepository;
    }
}
```

5.3.5.2 Example: `Logger`

Noncompliant Code Example

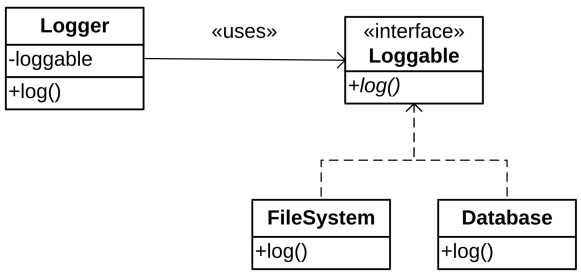
The noncompliant code example violates the Dependency Inversion Principle (DIP) by directly creating an instance of `FileSystem` within the `Logger` class. This creates a tight coupling between the `Logger` and the concrete implementation of `FileSystem`, making it difficult to change or extend the logging mechanism in the future.

Bad code

```
class Logger {
    private final FileSystem fileSystem;

    Logger() {
        this.fileSystem = new FileSystem();
    }

    public void log(String message) {
        this.fileSystem.log(message);
    }
}
```

This design violates the DIP because it directly depends on the concrete implementation `FileSystem`. This tight coupling makes it challenging to substitute or change the logging mechanism without modifying the `Logger` class.

Compliant Solution



The `Loggable` interface defines a contract for classes that can log messages.

Good code

```
public interface Loggable {
    void log(String message);
}
```

`FileSystem` and `Database`, that implement the `Loggable` interface encapsulate the specific logging mechanisms, such as writing to a file or a database.

Good code

```
class FileSystem implements Loggable {
    @Override
    public void log(String message) {
        // file handling and writing
    }
}
```

Good code

```
class Database implements Loggable {
    @Override
    public void log(String message) {
        // writing in database
    }
}
```

The `Logger` class is modified to accept an instance of `Loggable` through constructor injection.

Good code

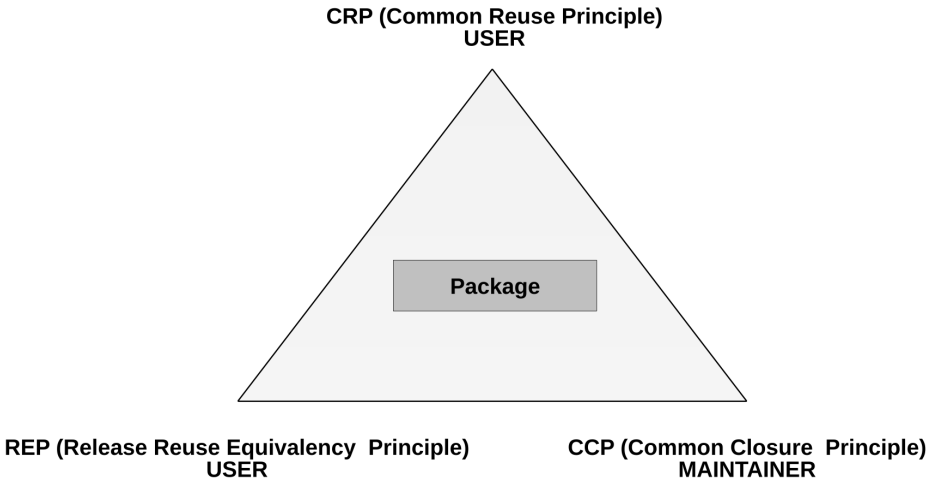
```
class Logger {
    private final Loggable loggable;

    Logger(Loggable loggable) {
        this.loggable = loggable;
    }

    public void log(String message) {
        this.loggable.log(message);
    }
}
```

By depending on the `Loggable` interface instead of a concrete implementation, the `Logger` class becomes decoupled from the specific logging mechanism. This allows different logging implementations to be provided at runtime without modifying the `Logger` class.

Good code

```
class Client {
    public static void main(String[] args) {
        Logger fsLogger = new Logger(new FileSystem());
        Logger dbLogger = new Logger(new Database());

        fsLogger.log("some text");
        dbLogger.log("other text");
    }
}
```

5.4 Packaging Principles - Cohesion



5.4.1 Release Reuse Equivalency Principle

The granule of reuse is the granule of release.

- The elements of reuse are the elements of the release.
- Classes that are reused together should be in the same package so that they can be released together.
- Reducing the number of modules to be distributed, which in turn reduces the effort of software distribution.
- Avoidance of version conflicts during reuse, which can only arise from the simultaneous release of several packages.
- The users of classes, which in turn depend on other classes, want to rely on a collective release of new versions.

5.4.2 Common Closure Principle

Classes that change together are packaged together.

- All classes in a package should be closed to the same type of changes according to the open closed principle.
- Corresponds to the single responsibility principle applied to packages.
- Changes to the requirements of the software that require changes to a class of a module should only affect the classes of the module.
- Compliance with this principle allows the software to be broken down into packages in such a way that future changes can be implemented in only a few modules.
- This ensures that changes to the software can be made without major side effects and thus relatively inexpensively.

5.4.3 Common Reuse Principle

Classes that are used together are packaged together.

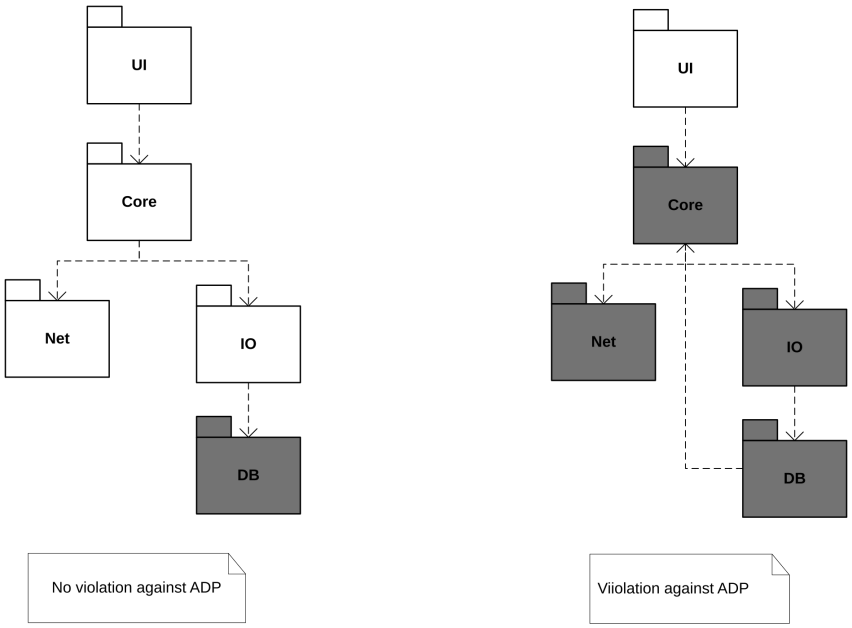
- Grouping of classes that are used together.
- Compliance with this principle ensures that units belonging together are subdivided.

5.5 Packaging Principles - Coupling

5.5.1 Acyclic Dependencies Principle

The dependency graph of packages must have no cycles.

- Classes and packages should not contain cyclic dependencies.
- Make testing, maintainability, reusability, builds, and releases more difficult if cycles are not eliminated.



Breaking up such cycles is always possible, but there are two possibilities:

If the dependency from package A to package B is to be inverted.

Use of the DIP:

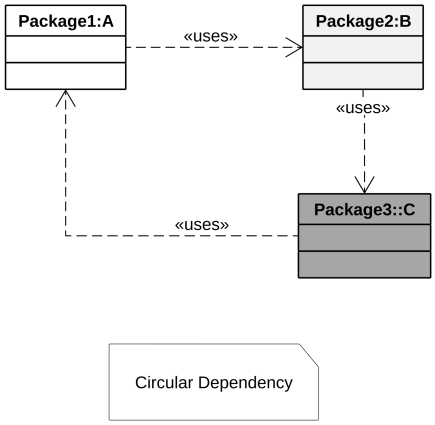
- Introduction of an interface in package A, which holds the methods required by B.
- Implementation of these interfaces in the corresponding classes of Package B.

Restructuring of the packages:

- Collecting all classes of a cycle in a separate package.
- Introducing one or more new packages with the classes on which the classes outside the cycle depend.

5.5.1.1 Example: `Cyclic dependency`

Noncompliant Code Example



The noncompliant code example violates the Acyclic Dependencies Principle (ADP) by creating a cyclic dependency between classes `A`, `B`, and `C`. This cyclic dependency makes the code difficult to understand, maintain, and test.

swcs.dp.adp.before.p1

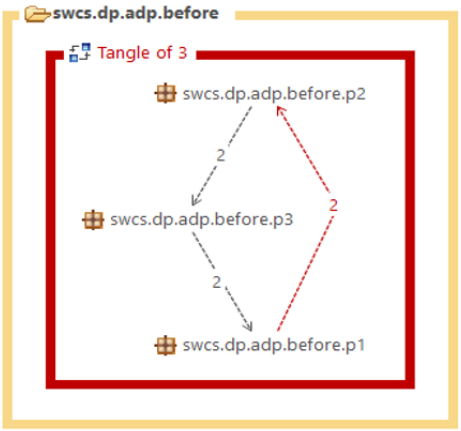
```
public class A {
    private final B b;

    public A(B b) {
        this.b = b;
    }
}
```

swcs.dp.adp.before.p2

```
public class B {
    private final C c;

    public B(C c) {
        this.c = c;
    }
}
```

swcs.dp.adp.before.p3

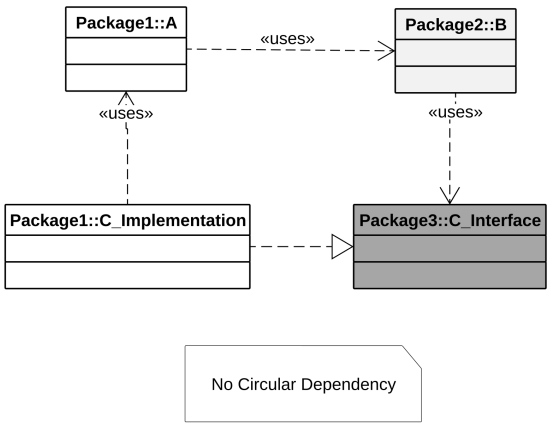
```
1  public class C {
2      private final A a;
3
4      public C(A a) {
5          this.a = a;
6      }
7  }
```



Violation against ADP

This design violates the ADP because it introduces cyclic dependencies. Class `A` depends on `B`, class `B` depends on `C`, and class `C` depends on `A`, creating a loop in the dependency graph.

Compliant Solution



The compliant solution resolves the cyclic dependencies by introducing an interface, `C`, to break the loop.

swcs.dp.adp.after.p1

```
public class A {
    private final B b;

    public A(B b) {
        this.b = b;
    }
}
```

Class `C` is refactored into an interface, and its implementation, `CImpl`, depends on class `A`.

swcs.dp.adp.after.p1

```
public class CImpl implements C {
    private final A a;

    public CImpl(A a) {
        this.a = a;
    }
}
```

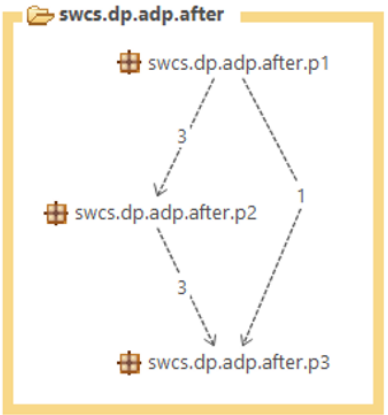
Now, class `CImpl` implements the `C` interface and receives an instance of `A` in its constructor. Class `A` and class `B` remain unchanged.

swcs.dp.adp.after.p2

```
public class B {
    private final C c;

    public B(C c) {
        this.c = c;
    }
}
```

swcs.dp.adp.after.p3

```
public interface C {
}
```



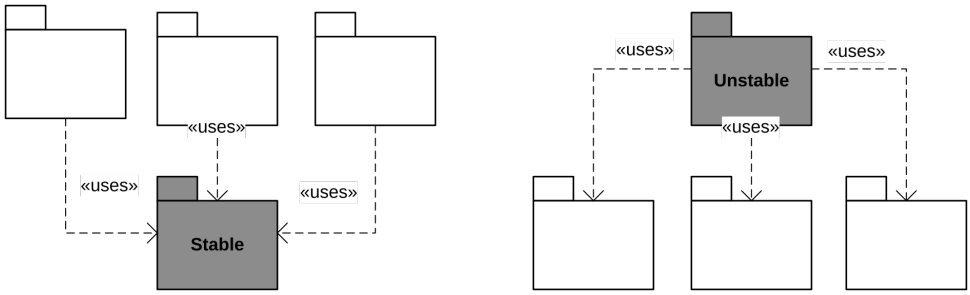
No violation against ADP

By introducing the `C` interface and its implementation `CImpl`, the cyclic dependencies are resolved. Class `A` depends on class `B`, class `B` depends on the `C` interface, and class `CImpl` depends on class `A`. This dependency structure creates a directed acyclic graph (DAG) rather than a cycle.

5.5.2 Stable Dependencies Principle

Depend on the direction of stability.

- The dependencies between packages should be in the same direction as the stability.
- A package should only depend on packages that are more stable than itself.
- The fewer dependencies a package has on others and the more dependencies other packages have on that package, the more stable it is.



Stable and unstable packages

Instability metric

The stability of a package is the ratio of outgoing coupling (`Ce`) to total coupling (`Ca + Ce`).

```
I = Instability, 0 ≤ I ≤ 1
I = 0, stable package
I = 1, unstable package
```

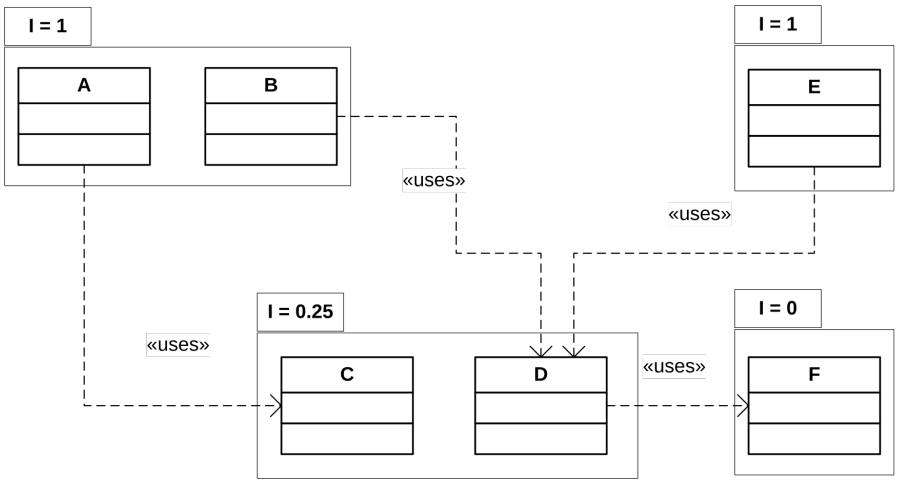
$$I = \frac{C_e}{C_a + C_e}$$

```
Ca = incoming dependencies (afferent couplings)
Ce = outgoing dependencies (efferent couplings)
```

An instability of 1 means that no other package depends on this package. This package depends on other packages and is as unstable as a package can be. Its lack of dependencies gives it no reason not to change. Furthermore, the packages it depends on can give it lots of reasons to change.

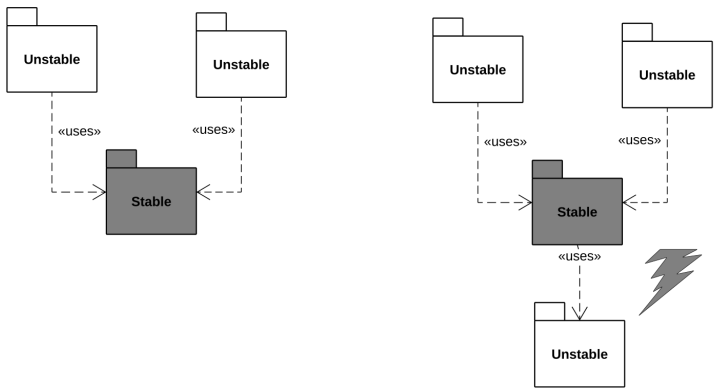
An instability of 0 means that the package depends on other packages, but does not depend on other packages at all. It is independent and is as stable as it can be. Its dependencies make it difficult to change. Further, it has no dependencies that could force it to change.

The SDP says that the metric of a package should be greater than the metrics of the packages it depends on. The metric of instability should decrease in the direction of dependency.



Instability

How do we achieve stability? The stable packages are not dependent on anything. A change from a dependent cannot spread to them and cause them to change. They are independent. Independent classes are classes that do not depend on anything else.
Another reason why a stable package gets stable is that many other classes depend on them. The more dependencies they have, the harder it is to make changes to them. If we change classes in the stable package, we would also have to change all the other classes that depend on them. So, there is a strong constraint that stops us from changing these classes. This fact not changing these classes increases their stability.



Stable package depend on unstable package

If all packages are maximally stable, the system would be unchangeable. This is not a desirable situation. We want to design our package structure so that some packages are unstable, and some are stable. The changeable packages should be at the top and depend on the stable packages at the bottom. If we place the unstable packages at the top and arrange them in this way, it does not violate the SDP.

5.5.3 Stable Abstractions Principles

Abstractness increases with stability.

- Establishes a relationship between stability and abstractness.
- Packages that are maximally stable should be maximally abstract.
- Unstable packages should be concrete.
- The abstractness of a package should be proportional to its stability.

Abstractness metric
The abstractness of a package is the ratio of abstract classes like abstract classes and interfaces in a package to the total number of classes in the package.

```
A = Abstractness, 0 ≤ A ≤ 1
A = 0, completely concrete package
A = 1, completely abstract package
```

$$A = \frac{N_a}{N_c}$$

```
Na = number of abstract classes, interfaces
Nc = total number of classes
```

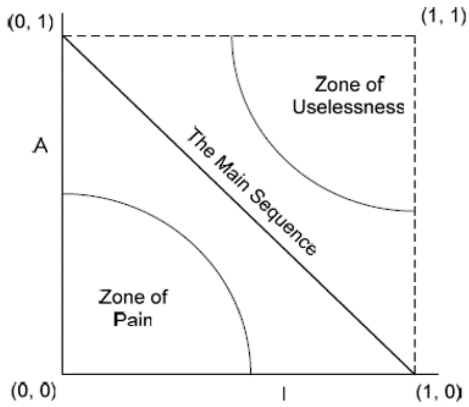
An abstractness of 0 means that the package has no abstract classes. An abstractness of 1 means that the package contains nothing but abstract classes such as interfaces and abstract classes.

Relationship between Instability (I) and Abstractness (A)

The relationship between I and A can be defined within a graph with A on the vertical axis and I on the horizontal axis. There are two *good* types of packages on this graph, the first **Interface packages** that are maximally stable and abstract at the top left at (0,1). The second is the **Implementation packages** that are maximally unstable, and concrete is at the bottom right at (1,0).

However, not all packages can be classified in one of these two categories. Thus, packages have degrees of abstraction and stability. Therefore, we must accept that there is a place of points on the A/I graph that defines meaningful positions for packages. This can be done with the Main Sequence which is a line from (0,1) to (1,0). This line represents packages whose abstractness is *balanced* with stability. A packet on the Main Sequence is neither *too abstract* for its stability, nor is it *too unstable* for its abstractness. It has the *right* number of concrete and abstract classes in proportion to its efferent and afferent dependencies.

We can infer in which areas no packages should be, the **Zone of Pain** and the **Zone of Uselessness**.



The Main Sequence

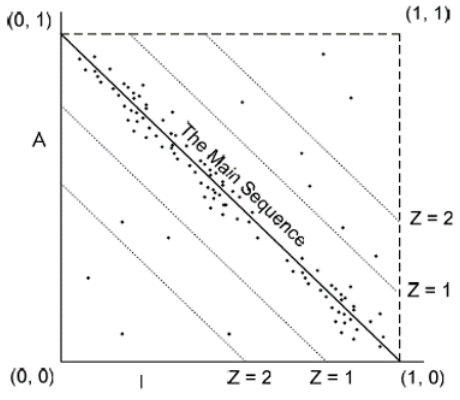
- Ideal line
- Even better when package at (0.1) or (1.0), hardly implemented in practice

Zone of Pain

- Stable non-abstract classes
- Difficult to change because stable, difficult to extend because not abstract
- Exceptions are utility classes such as `String`

Zone of Uselessness

- Unstable very abstract classes
- Are abstract but are nowhere extended



Distance from the Main Sequence

The most desirable positions for a package are at one of the two endpoints of the Main Sequence. However, most packages do not have such ideal attributes. The other packages have the best characteristics when they are on or near the ideal line, which can be defined with the distance to Main Sequence. For each package, the distance to the ideal line between maximum stability

and abstractness and maximum instability and concreteness can be calculated. The greater the distance, the worse the SAP is fulfilled.

$$D = \left| \frac{\mathrm{A} + \mathrm{I} - 1}{\sqrt{2}} \right|$$

or normalized

$$D' = \left| \mathrm{A} + \mathrm{I} - 1 \right|$$

D = Distance from the ideal line, 0 ≤ D ≤ 0.707

D'= Normalised distance from the ideal line, 0 ≤ D' ≤ 1

A = Abstractness of a package

I = Instability of a package

5.6 Further Design Principles

5.6.1 Speaking Code Principle

The Speaking Code principle is a design principle that says that code should be written so that it is easy to read and understand. This is important because code is often read more than it is written. When we write code, we're communicating with computers, but when we read code, we're communicating with other humans. That's why it's important to make sure our code is easy to read. So following this principle can also make it easier to work with other people on code projects.

- Code should **communicate** its purpose, even without comment and documentation.
- Comments are no substitute for bad code.
- Clear and expressive code with few comments is much better than messy and complex code with numerous comments.
- Instead of spending time writing comments that explain the chaos, spend time cleaning up the chaos.
- Comment must be adjusted when the code is changed.
- Using of whitespaces and consistent formatting.

Bad code

```
1 if ((bike.type & TYP_FLAG) && bike.status == READY) { ... }
```

Good code

```
1 if (bike.isRentable()) { ... }
```

5.6.2 Keep It Simple (and) Stupid!

No one is in a position to manage large complexity. Not in software, not in any other aspect of life. Complexity is the killer of good software and therefore simplicity is the enabler. Simplicity is something that takes a lot of time and practice to recognize and produce. Of course, many rules can be followed.

An example would be to have methods with only a few parameters. In the best case, none or only one. We can write `"Keep It Simple (and) Stupid!".contains("Simple")` and without reading any documentation, everyone will immediately understand what this does and why. The method does one thing and not more. There is no complicated context and no other arguments that can be passed to the method. There are no *special cases* or any caveats.

- Means that always the **simplest** solution of a problem should be chosen.
- Avoid code that is too complex and therefore too complicated.
- Finding simple solutions is a rule that helps to avoid various errors.
- A healthy rejection of non-simple solutions is a sign of searching for good code.
- The more difficult code is to explain, the more likely it is that it is more complicated than necessary and is not the most elegant solution.
- Make things as simple as possible, but not simpler!

5.6.3 Don’t Repeat Yourself / Once and Only Once

Don’t Repeat Yourself (DRY) is a principle of software development aimed at reducing repetition of software development patterns, replacing it with abstractions or using data structures. When the DRY principle is applied successfully, a modification to any one element of a system does not require a similar change to other logically unrelated elements.

Once and Only Once (OOO) is a similar but more specific software development principle which states that for each element of software, there should be only one single, unambiguous representation within the system. This can be accomplished by ensuring that code is not duplicated, and by abstracting code into modules or functions that can be reused.

- Refers to the avoidance of **duplicated code**, i.e. code fragments that are implemented in the same or very similar way in several places.
- Redundant existing source code is difficult to maintain, as consistency between the individual duplicates must be ensured.
- In systems that remain loyal to the DRY principle, however, changes need only be made in one place.

As a programmer, it is important to avoid repeating yourself. This not only saves time, but also makes your code more reliable and easier to maintain. When you need to make a change, you only have to do it in one place, and the change will propagate throughout your code. This is in contrast to Write Everything Twice (WET) programming, in which you write code that is duplicated, and you have to make changes in multiple places. DRY programming is a more efficient and effective way to write code, and it is something that all programmers should strive for.

5.6.4 You Ain’t Gonna Need It!

You Ain’t Gonna Need It, also known as YAGNI, is a programming principle that suggests that programmers should not add features or functionality to a software system unless it is absolutely necessary.

Resist the temptation to add features or functionality that you think might be needed in the future, and instead focus on building only what is needed right now. This may seem like a no-brainer, but in practice it can be difficult to resist the temptation to add “just in case” features.

YAGNI can help to keep code clean and maintainable, and can also help to reduce development time and costs.

- Where no requirement, also no implementation!
- A program should only implement functionality when this functionality is needed.
- Contrary to this approach, in practice it is often attempted to prepare programs for possible future change requests and features through additional or more general code. *“…this is what THEY will demand soon anyway…”*
- Such code can be very annoying to read, understand, change existing functionality, and costs time and money!

5.6.5 Separation Of Concerns

Separation of Concerns (SoC) is a principle to relieve the complexity of a program by dividing it into distinct parts, each with a separate concern.

When done effectively, separation of concerns can make code easier to maintain and improve over time. This is because each concern can be isolated and changed without affecting the others. This can make code more modular and reusable, and less error-prone.

Overall, separation of concerns programming is a good thing. It can help you keep your code organized and make it easier to maintain.

- A component should have exactly one task.
- Avoid mixing of several responsibilities in e.g. one class.
- Components become simpler, easier to understand, maintain and have better reusability.
- The complex tasks of a unit should be self-contained (**high cohesion**).
- The unit should depend as little as possible on other units (**low coupling**).

6. Design Patterns of the Gang of Four

Each pattern describes a problem that occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way Twice.
–Christopher Alexander 1977, A Pattern Language: Towns, Buildings, Construction

- Design patterns originally come from architecture.
- 1977 Christoph Alexander describes in his book *A Pattern Language. Towns, Buildings, Construction* the design patterns for landscape and urban planning, among other things.
- Kent and Cunningham drew their inspiration for design patterns in software development from Alexander (about 1987).
- Gamma, Vlissides, Johnson, and Helm transferred the ideas to software technology and designed a pattern catalog.
- The Gang of Four (GoF) published the book "Design Patterns" in 1995 and made design patterns popular.

Design patterns are solution patterns for recurring design problems in software design. There a based-on experience, they are often only understandable and comprehensible with appropriate background knowledge and personal experience. They establish communication by forming a vocabulary (Pattern Language) that developers use to discuss design. The basics of object-oriented design like *Encapsulation*, *Responsibility*, *Polymorphism* and *Inheritance* must be understood.

Why are design patterns important?

Many problems repeat themselves in software development. Class hierarchies become inflexible over time and individual classes grow uncontrollably. New extensions are very difficult to implement, and the dependencies of a class increase so that the function can be fulfilled. The encapsulation of classes is becoming softened and more and more of the internal state is being exposed.

The wheel does not have to be reinvented every time, at least the schematic solutions are well documented. Since design patterns have existed for a long time, the solutions have been proven many times and the consequences are well understood. Furthermore, software design can be discussed on a higher level of abstraction and thus more efficiently since the wording is already self-explanatory and known to the developers. Instead of looking for the solution to a problem, there is a sense of recognizing connections that suggest the use of a design pattern.

Classification of design patterns according to GoF

The 23 design patterns of the Gang of Four are divided into three groups. The categories describe the intention that is to be achieved with a design pattern.

Creational Patterns

How objects or even class hierarchies can be created?

Group	Pattern	Description
Creational	Singleton	Ensure a class only has one instance, and provide a global point of access to it.
	Builder	Separate the construction of a complex object from its representation so that the same construction process can create different representations.
	Factory Method	Define an interface for creating an object, but let the subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.
	Abstract Factory	Provide an interface for creating families of related or dependent objects without specifying their concrete class.
	Prototype	Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.

Structural Patterns

What structure must a composition of classes and objects have to solve a certain problem?

Group	Pattern	Description
Structural	Facade	Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.
	Decorator	Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.
	Adapter	Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn‘t otherwise because of incompatible interfaces.
	Composite	Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.
	Bridge	Decouple an abstraction from its implementation so that the two can vary independently.
	Flyweight	Use sharing to support large numbers of fine-grained objects efficiently.
	Proxy	Provide a surrogate or placeholder for another object to control access to it.

Behavioural Patterns

How do the objects work together at runtime? How is responsibility distributed among them?

Group	Pattern	Description
Behavioral	State	Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.
	Template Method	Define a skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm‘s structure.
	Strategy	Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

Group	Pattern	Description
	Observer	Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
	Chain of Responsibility	Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.
	Command	Encapsulates a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.
	Interpreter	Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.
	Iterator	Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.
	Mediator	Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.
	Memento	Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.
	Visitor	Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

With the introduction of lambdas in Java 8 some design patterns can be implemented in a different way[1] with lambdas.

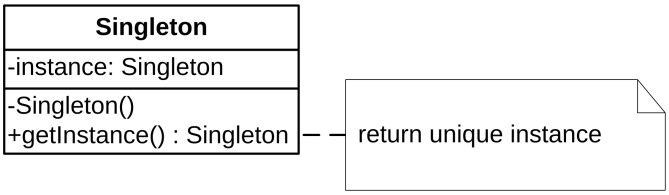
[1] https://github.com/mariofusco/from-gof-to-lambda

6.1 Creational

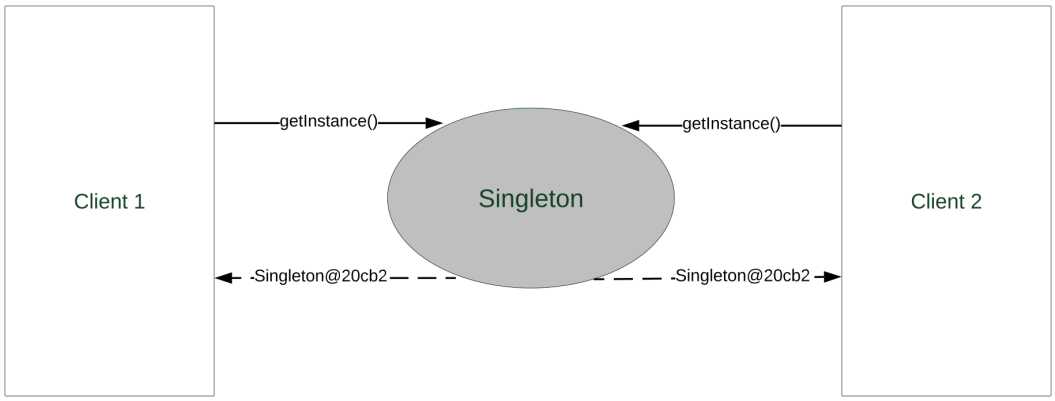
6.1.1 Singleton

Ensure a class only has one instance, an provide a global point of access to it.

- The Singleton design pattern ensures that there is only **one** instance of a class.
- There are three ways how singletons can be implemented.



Singletons are not created at each request. Only single instance is reused again and again.



6.1.1.1 Example: `Lazy loading`

Lazy loading

```
public final class Singleton {
    private static Singleton instance;

    private Singleton() {}

    public static synchronized Singleton getInstance() {
      if (instance == null) {
        instance = new Singleton();
      }

      return instance;
    }
}
```

6.1.1.2 Example: `Eager loading`

Eager loading

```
public final class Singleton {
    private static final Singleton instance = new Singleton();

    private Singleton() {}

    public static Singleton getInstance() {
        return instance;
    }
}
```

6.1.1.3 Example: `Enum singleton`

This approach provides serialization for free and offers a guarantee against multiple instantiations. A single-element enum type is the best way to implement a singleton.[2]

Enum singleton, is the preferred approach

```
public enum Singleton {
    INSTANCE
}
```

Advantages:

- **Easy to use**. A singleton class is written quickly and easily.
- Guarantee that only **one** instance exists in a JVM at a time.
- Compared to global variables there are several advantages:
  - **Access control**. The Singleton encapsulates its creation and can therefore control exactly when and how access to the Singleton is allowed.
  - **Lazy-loading**. Singletons can only be created when they are needed.

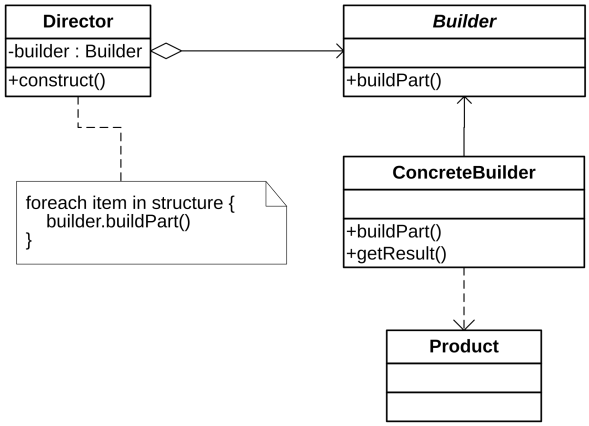
Disadvantages:

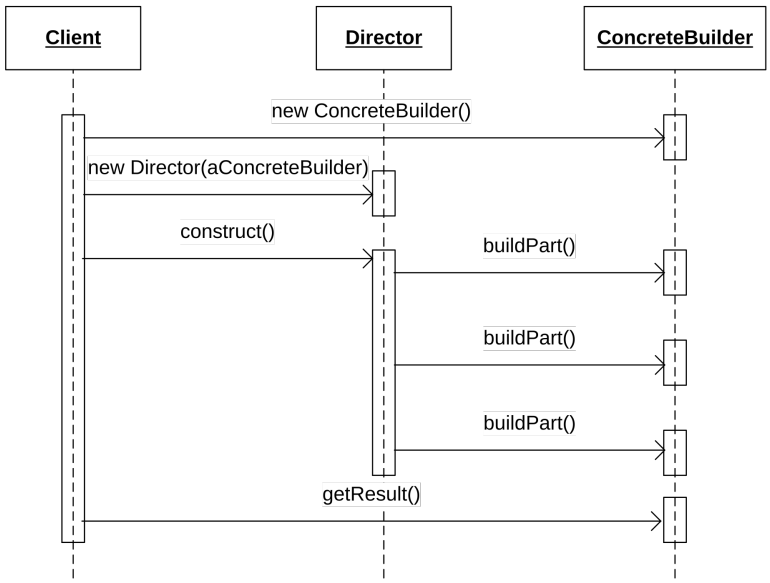
- **Problematic destruction**. To destroy objects in languages with garbage collection, an object must no longer be referenced. This is difficult to ensure with singletons. Due to the global availability, it happens very quickly that code parts still hold a reference to the singleton.
- Especially in multi-user applications, a Singleton can reduce the **performance** because it represents a bottleneck - especially in the synchronized form.
- **Uniqueness beyond physical limits**. Ensuring the uniqueness of a Singleton across physical boundaries (JVM) is difficult.

[2] Bloch, Joshua: Effective Java – Third Edition, Item 3

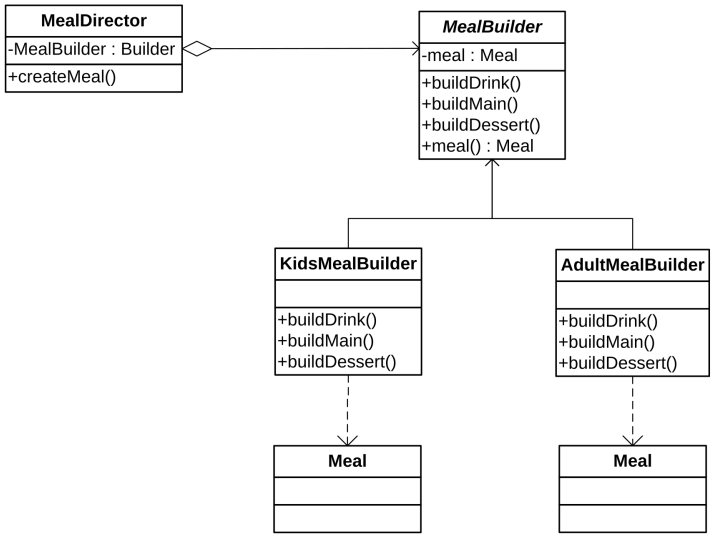
6.1.2 Builder

Separate the construction of a complex object from its representation so that the same construction process can create different representations.





6.1.2.1 Example: `MealBuilder`



MealBuilder

```
1  public abstract class MealBuilder {
2      private final Meal meal;
3
4      protected MealBuilder() {
5          this.meal = new Meal();
6      }
7
8      public abstract void buildDrink();
9      public abstract void buildMain();
10     public abstract void buildDessert();
11
12     public Meal meal() {
13         return this.meal;
14     }
15 }
```

KidsMealBuilder

```
1  public class KidsMealBuilder extends MealBuilder {
2      public void buildDrink() {
3          // add drinks to the meal
4      }
5
6      public void buildMain() {
7          // add main part of the meal
8      }
9
10     public void buildDessert() {
11         // add dessert part to the meal
12     }
13 }
```

AdultMealBuilder

```
public class AdultMealBuilder extends MealBuilder {
    public void buildDrink() {
        // add drinks to the meal
    }

    public void buildMain() {
        // add main part of the meal
    }

    public void buildDessert() {
        // add dessert part to the meal
    }
}
```

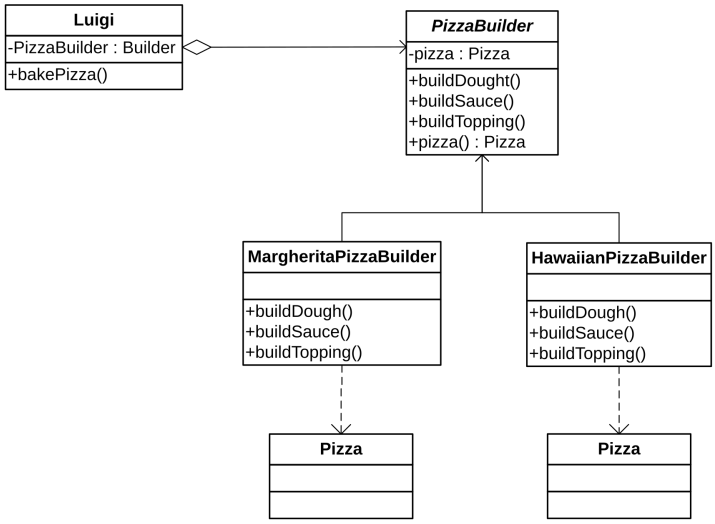
MealDirector

```
public class MealDirector {

    public Meal createMeal(MealBuilder builder) {
        builder.buildDrink();
        builder.buildMain();
        builder.buildDessert();

        return builder.meal();
    }
}
```

Client

```
class Client {

    public static void main(String[] args) {
        MealDirector director = new MealDirector();
        MealBuilder builder = null;
        boolean isKid = true;

        if (isKid) {
            builder = new KidsMealBuilder();
        } else {
            builder = new AdultMealBuilder();
        }

        Meal meal = director.createMeal(builder);
    }
}
```

6.1.2.2 Example: `PizzaBuilder`



PizzaBuilder

```
abstract class PizzaBuilder {
    private final Pizza pizza;

    PizzaBuilder() {
        this.pizza = new Pizza();
    }

    public Pizza pizza() {
        return this.pizza;
    }

    public abstract void buildDough();

    public abstract void buildSauce();

    public abstract void buildTopping();
}
```

HawaiianPizzaBuilder

```
class HawaiianPizzaBuilder extends PizzaBuilder {

    @Override
    public void buildDough() {
        this.pizza().dough("cross");
    }

    @Override
    public void buildSauce() {
        this.pizza().sauce("mild");
    }

    @Override
    public void buildTopping() {
```

```
15          this.pizza().topping("ham+pineapple");
16      }
17  }
```

MargheritaPizzaBuilder

```
1   class MargheritaPizzaBuilder extends PizzaBuilder {
2
3       @Override
4       public void buildDough() {
5           this.pizza().dough("cross");
6       }
7
8       @Override
9       public void buildSauce() {
10          this.pizza().sauce("mild");
11      }
12
13      @Override
14      public void buildTopping() {
15      }
16  }
```

Pizza

```
1   final class Pizza {
2       private String dough;
3       private String sauce;
4       private String topping;
5
6       public void dough(String dough) {
7           this.dough = dough;
8       }
9
10      public void sauce(String sauce) {
11          this.sauce = sauce;
12      }
13
14      public void topping(String topping) {
15          this.topping = topping;
16      }
17  }
```

Luigi

```
class Luigi {
    private final PizzaBuilder pizzaBuilder;

    Luigi(PizzaBuilder pizzaBuilder) {
        this.pizzaBuilder = pizzaBuilder;
    }

    public Pizza bakePizza() {
        this.pizzaBuilder.buildDough();
        this.pizzaBuilder.buildSauce();
        this.pizzaBuilder.buildTopping();

        return this.pizzaBuilder.pizza();
    }
}
```

Client

```
class Client {

    public static void main(String[] args) {
        Luigi luigi = new Luigi(new HawaiianPizzaBuilder());
        Pizza pizza = luigi.bakePizza();
    }
}
```

6.1.2.3 Example: `Email`

Email

```
public final class Email {

    private final String subject;
    private final String message;
    private final String recipients;

    private Email(String subject, String message, String recipients) {
        this.subject = subject;
        this.message = message;
        this.recipients = recipients;
    }

    public String subject() {
        return this.subject;
    }

    public String message() {
        return this.message;
    }

    public String recipients() {
        return this.recipients;
    }

```

```
25      public void send() {
26      }
27
28      public static Builder builder() {
29          return new Builder();
30      }
31
32      public static final class Builder {
33
34          private String subject;
35          private String message;
36          private String signature;
37          private final Set<String> recipients;
38
39          public Builder() {
40              this.recipients = new HashSet<>();
41          }
42
43          public Builder withSubject(String subject) {
44              this.subject = subject;
45              return this;
46          }
47
48          public Builder withMessage(String message) {
49              this.message = message;
50              return this;
51          }
52
53          public Builder withSignature(String signature) {
54              this.signature = signature;
55              return this;
56          }
57
58          public Builder addRecipient(String recipient) {
59              this.recipients.add(recipient);
60              return this;
61          }
62
63          public Builder removeRecipient(String recipient) {
64              this.recipients.remove(recipient);
65              return this;
66          }
67
68          public Email build() {
69              return new Email(this.subject,
70                      String.format("%s%n%s", this.message, this.signature),
71                      String.join(".", this.recipients));
72          }
73      }
74  }
```

Client

```
class Client {

    public static void main(String[] args) {
        Email email = Email.builder()
                .addRecipient("bad@foo.com")
                .addRecipient("coder@foo.com")
                .withMessage("Your first Builder Pattern")
                .withSignature("Clean Coder")
                .build();

        email.send();
    }
}
```

6.1.2.4 Example: `ImmutablePerson`

ImmutablePerson

```
public final class ImmutablePerson {

    private final String name;
    private final String city;
    private final List<String> favoriteDishes;

    private ImmutablePerson(Builder builder) {
        this.name = builder.name;
        this.city = builder.city;
        this.favoriteDishes = builder.favoriteDishes;
    }

    public String name() {
        return this.name;
    }

    public String city() {
        return this.city;
    }

    public List<String> favoriteDishes() {
        return this.favoriteDishes != null ? new ArrayList<>(this.favoriteDishes) : null;
    }

    public Builder toBuilder() {
        return new Builder(this);
    }

    public static Builder builder() {
        return new Builder();
    }

    public static final class Builder {

        private String name;
        private String city;
        private final List<String> favoriteDishes;
```

```
38
39          public Builder() {
40              this.favoriteDishes = new ArrayList<>();
41          }
42
43          public Builder(ImmutablePerson person) {
44              this.name = person.name;
45              this.city = person.city;
46              this.favoriteDishes = person.favoriteDishes != null
47                  ? new ArrayList<>(person.favoriteDishes) : null;
48          }
49
50          public Builder withName(String name) {
51              this.name = name;
52              return this;
53          }
54
55          public Builder withCity(String city) {
56              this.city = city;
57              return this;
58          }
59
60          public Builder addFavoriteDish(String dish) {
61              this.favoriteDishes.add(dish);
62              return this;
63          }
64
65          public Builder removeFavoriteDish(String dish) {
66              this.favoriteDishes.remove(dish);
67              return this;
68          }
69
70          public ImmutablePerson build() {
71              return new ImmutablePerson(this);
72          }
73      }
74  }
```

Client

```
1   class Client {
2
3       public static void main(String[] args) {
4           ImmutablePerson hugo = ImmutablePerson.builder()
5                   .withName("Hugo")
6                   .withCity("Nuremberg")
7                   .addFavoriteDish("Pasta")
8                   .addFavoriteDish("Pancake")
9                   .build();
10
11          ImmutablePerson mia = ImmutablePerson.builder()
12                  .withName("Mia")
13                  .withCity("Nuremberg")
14                  .addFavoriteDish("Sausage")
15                  .build();
16
17          ImmutablePerson hugoAfterMove = hugo.toBuilder()
```

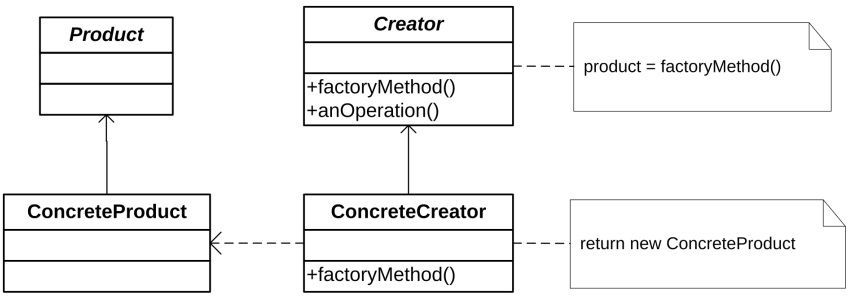
```
18                  .withCity("Munich")
19                  .build();
20
21          ImmutablePerson miaAfterMove = mia.toBuilder()
22                  .withCity("Munich")
23                  .build();
24      }
25  }
```

Advantages:

- **Different expressions** of complex products can be built.
- Further expressions can be easily added with **new concrete builder classes**.
- The logic for the construction is **separated**.
- A client does not need to know anything about the **generation process**.
- **Fine-grained control** over the generation process is possible. The product is built piece by piece under the director's control. Other generation patterns build the products in one piece.

6.1.3 Factory Method

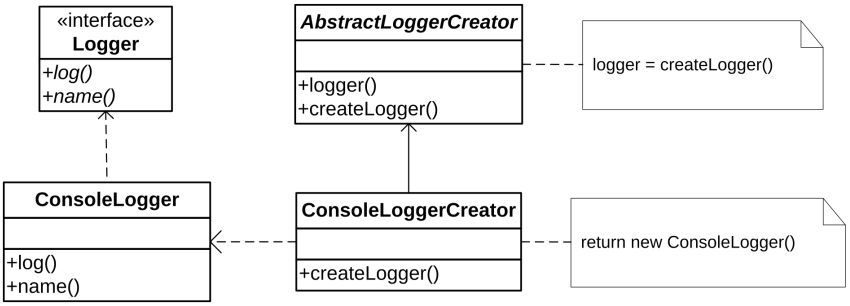
Define an interface for creating an object, but let the subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.



Use cases:

- Separation of object processing (**How?**) from concrete object creation (**What?**).
- Delegation of object instantiation to subclass.
- Cases in which an increasing number and shape of products can be expected. As well as scenarios in which all products have to go through a general manufacturing process, no matter what kind of product they are.
- If the products to be created are not known or should not be defined from the beginning.

6.1.3.1 Example: `Logger`



Logger

```
1  public interface Logger {
2      void log(String message);
3      String name();
4  }
```

ConsoleLogger

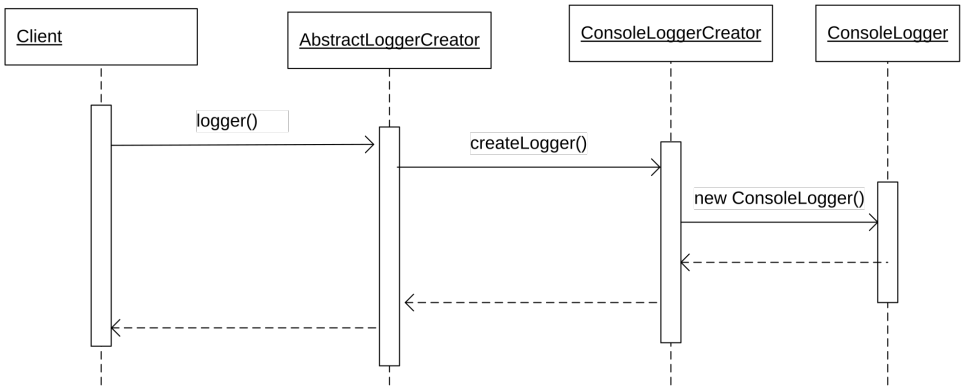
```java
public class ConsoleLogger implements Logger {

    @Override
    public void log(String message) {
        System.err.println(message);
    }

    @Override
    public String name() {
        return getClass().getSimpleName();
    }
}
```

AbstractLoggerCreator

```java
public abstract class AbstractLoggerCreator {

    public Logger logger() {
        // depending on the subclass, we'll get a particular logger.
        Logger logger = createLogger();

        // could do other operations on the logger here
        logger.log(String.format("Logger %s are used.", logger.name()));

        return logger;
    }

    protected abstract Logger createLogger();
}
```

ConsoleLoggerCreator

```java
public class ConsoleLoggerCreator extends AbstractLoggerCreator {

    @Override
    public Logger createLogger() {
        return new ConsoleLogger();
    }
}
```



Client

```
class Client {

    public static void main(String[] args) {
        AbstractLoggerCreator creator = new ConsoleLoggerCreator();
        Logger logger = creator.logger();
        logger.log("message");
    }
}
```

6.1.3.2 Example: `Department`

Employee

```
public interface Employee {
    void paySalary();
    void dismiss();
}
```

Programmer

```
public class Programmer implements Employee {

    private final int id;

    public Programmer(int id) {
        this.id = id;
    }

    @Override
    public void paySalary() {
    }

    @Override
    public void dismiss() {
    }

}
```

MarketingSpecialist

```
public class MarketingSpecialist implements Employee {

    private final int id;

    public MarketingSpecialist(int id) {
        this.id = id;
    }

    @Override
    public void paySalary() {
    }

    @Override
    public void dismiss() {
    }
}
```

Department

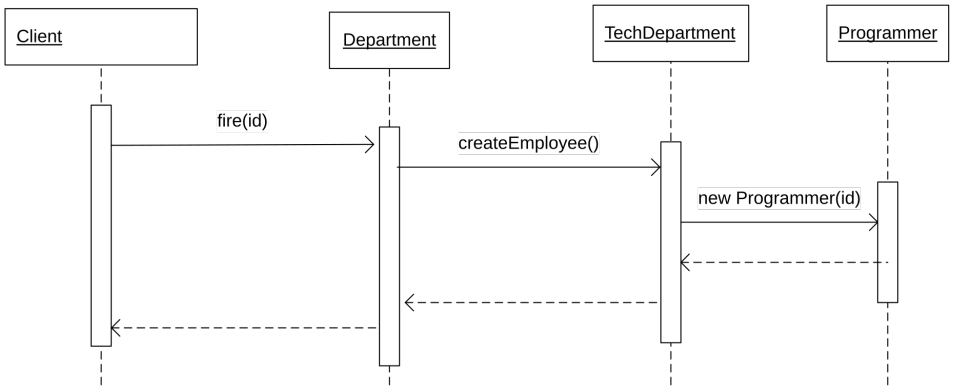
```
public abstract class Department {

    protected abstract Employee createEmployee(int id);

    public void fire(int id) {
        Employee employee = createEmployee(id);

        employee.paySalary();
        employee.dismiss();
    }
}
```

TechDepartment

```
public class TechDepartment extends Department {

    @Override
    public Employee createEmployee(int id) {
        return new Programmer(id);
    }
}
```

MarketingDepartment

```
public class MarketingDepartment extends Department {

    @Override
    public Employee createEmployee(int id) {
        return new MarketingSpecialist(id);
    }
}
```



Client

```
class Client {

    public static void main(String[] args) {
        Department department = new TechDepartment();
        department.fire(1);
    }
}
```

Advantages:

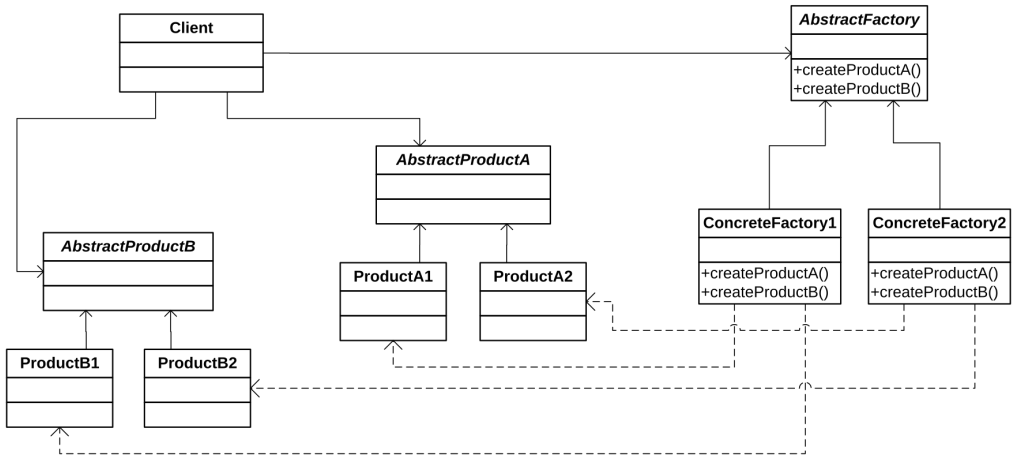
- Manufacturing process is **separated** from a concrete implementation.
- Different product implementations can go through the **same** production process.
- Reusability and consistency:
  - Encapsulation of the object creation code in its class. This creates a uniform and central interface for the client. The product implementation is **decoupled** from its use.

Disadvantages:

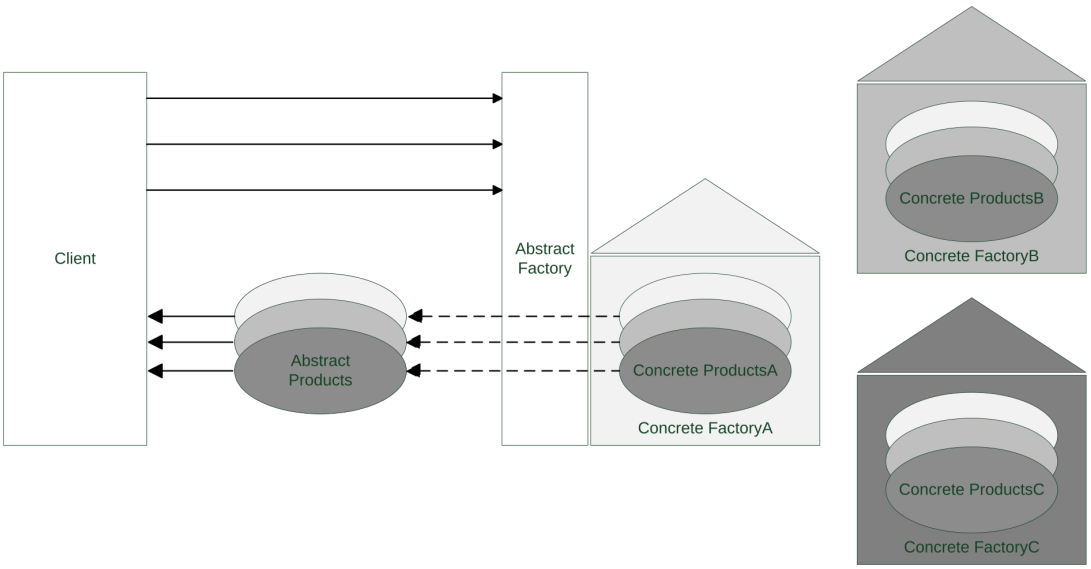
- If many simple objects are to be created, the effort is **disproportionately high**, because the creator always has to be derived.

6.1.4 Abstract Factory

Provide an interface for creating families of related or dependent objects without specifying their concrete class.



The Abstract Factory Design Pattern serves to define a **coherent family** of products.

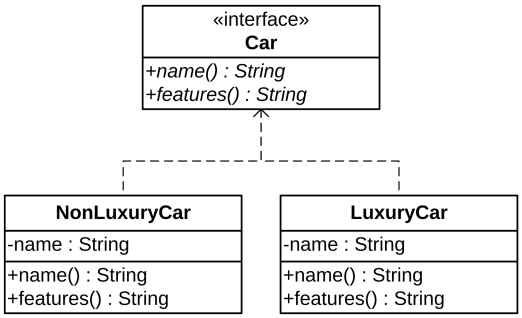


Use cases:

- When different objects are created for a context and therefore always have to be created coherently.
- When a system must be configured with different sets of objects or should be able to do so.
- When a system should be independent of how certain objects are created.
- If a family of objects is provided but no statements can or should be made about the concrete implementations. Instead, interfaces are provided.
- Typical applications of abstract factors are e.g., GUI libraries that support different Look & Feels.

6.1.4.1 Example: `Car`

Product family	Classes
Luxury	**LuxuryCar**, LuxurySUV
Non luxury	**NonLuxuryCar**, NonLuxurySUV



Car

```
public interface Car {
    String name();
    String features();
}
```

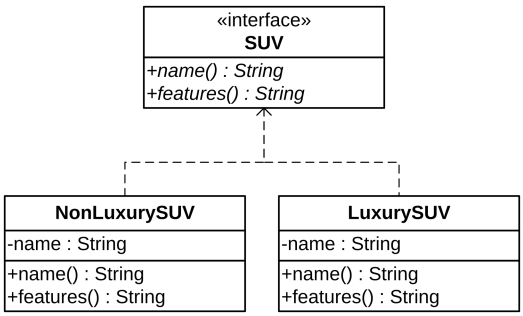
NonLuxuryCar

```
public class NonLuxuryCar implements Car {
    private final String name;

    public NonLuxuryCar(String name) {
        this.name = name;
    }

    @Override
    public String name() {
        return this.name;
    }

    @Override
    public String features() {
        return "Non-Luxury Car Features ";
    }
}
```

LuxuryCar

```java
public class LuxuryCar implements Car {
    private final String name;

    public LuxuryCar(String name) {
        this.name = name;
    }

    @Override
    public String name() {
        return this.name;
    }

    @Override
    public String features() {
        return "Luxury Car Features ";
    }
}
```

Product family	Classes
Luxury	LuxuryCar, **LuxurySUV**
Non luxury	NonLuxuryCar, **NonLuxurySUV**



Car

```java
public interface SUV {
    String name();
    String features();
}
```

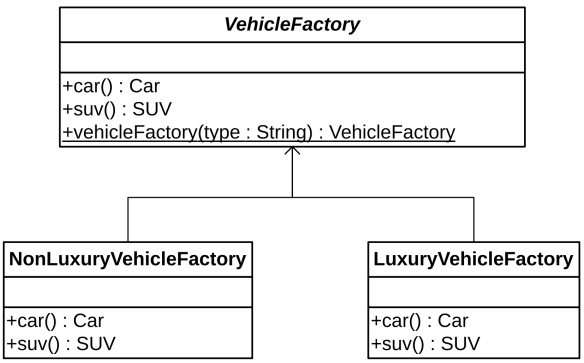
NonLuxurySUV

```java
public class NonLuxurySUV implements SUV {
    private final String name;

    public NonLuxurySUV(String name) {
        this.name = name;
    }

    @Override
    public String name() {
        return this.name;
    }
```

```
12
13      @Override
14      public String features() {
15          return "Non-Luxury SUV Features ";
16      }
17  }
```

LuxurySUV

```
1   public class LuxurySUV implements SUV {
2       private final String name;
3
4       public LuxurySUV(String name) {
5           this.name = name;
6       }
7
8       @Override
9       public String name() {
10          return this.name;
11      }
12
13      @Override
14      public String features() {
15          return "Luxury SUV Features ";
16      }
17  }
```



VehicleFactory

```
1   public abstract class VehicleFactory {
2       public static final String LUXURY_VEHICLE = "Luxury";
3       public static final String NON_LUXURY_VEHICLE = "Non-Luxury";
4
5       public abstract Car car();
6       public abstract SUV suv();
7
8       public static VehicleFactory vehicleFactory(String type) {
9           if (VehicleFactory.LUXURY_VEHICLE.equals(type)) {
10              return new LuxuryVehicleFactory();
11          }
12          if (VehicleFactory.NON_LUXURY_VEHICLE.equals(type)) {
13              return new NonLuxuryVehicleFactory();
14          }
```

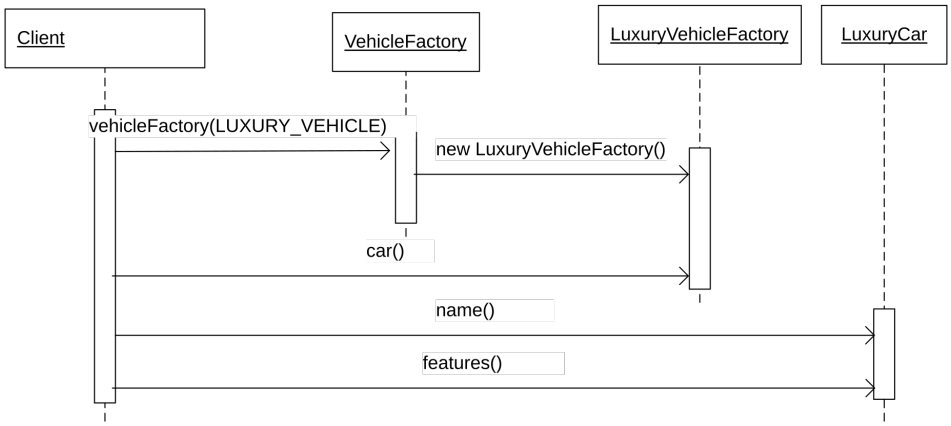
```
15
16          return new LuxuryVehicleFactory();
17      }
18  }
```

NonLuxuryVehicleFactory

```
1   public class NonLuxuryVehicleFactory extends VehicleFactory {
2
3       @Override
4       public Car car() {
5           return new NonLuxuryCar("NL-C");
6       }
7
8       @Override
9       public SUV suv() {
10          return new NonLuxurySUV("NL-S");
11      }
12  }
```

LuxuryVehicleFactory

```
1   public class LuxuryVehicleFactory extends VehicleFactory {
2
3       @Override
4       public Car car() {
5           return new LuxuryCar("L-C");
6       }
7
8       @Override
9       public SUV suv() {
10          return new LuxurySUV("L-S");
11      }
12  }
```



Client

```
class Client {

    public static void main(String[] args) {
        VehicleFactory vf = VehicleFactory.vehicleFactory(VehicleFactory.LUXURY_VEHICLE);

        Car car = vf.car();
        System.out.println("Name: " + car.name() + "  Features: " + car.features());

        SUV suv = vf.suv();
        System.out.println("Name: " + suv.name() + "  Features: " + suv.features());
    }
}
```

Advantages:

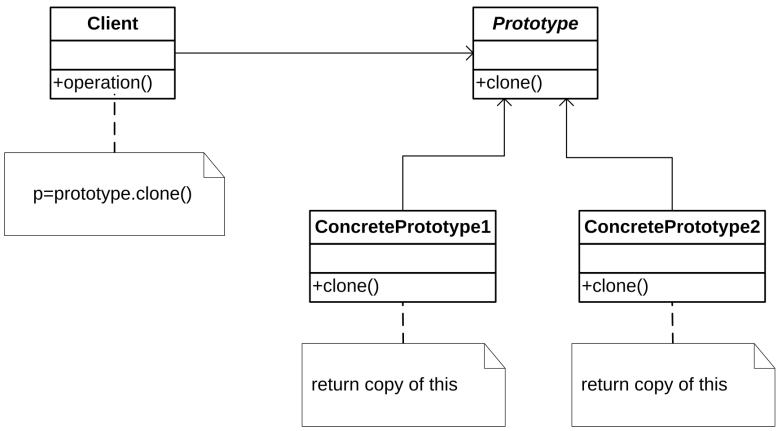
- By **shielding the concrete classes**, the client code becomes generally valid. No code is necessary for special cases.
- **Consistency**. It is ensured that only those objects that fit together reach the client.
- **Flexibility**. Entire object families can be exchanged since the client only relies on abstractions (Abstract Factory, product interfaces).
- **Simple extension with new product families**. New product families can be integrated into the system very easily. All that is required is the reimplementation of the factory interface. Afterward, the new factory only needs to be instantiated at a central point in the client.

Disadvantages:

- **Inflexibility regarding new family members**. If a new product is to be added to the product family, the interface of the Abstract Factory must be changed. However, this leads to the breaking of the code of all concrete factors. The change effort is high.
- A new implementation with great similarity to an existing one still requires a new factory.

6.1.5 Prototype

Specify the kinds of objects to create using a prototypical instance and create new objects by copying this prototype.



Use cases:

- Creating a copy of a prototype object and making required modifications.

Java provides some ways to copy an object to create a new one. One is using the `java.lang.Cloneable` marker interface. You need to implement the `java.lang.Cloneable` interface and override the `clone` method which every object inherits from the `Object` class.

There are two categories regarding cloning, **shallow** and **deep cloning**.

A shallow copy copies all reference types or value types, but it does not copy the objects that the references refer to. The references in the new object point to the same objects that the references in the original object point to. The default `clone` method implementation in Java creates a clone as a shallow copy.

A deep copy copies the elements, and everything directly or indirectly referenced by the elements.

6.1.5.1 Example: `Person - Shallow copy`

Person

```
1  public class Person implements Cloneable {
2
3      private String name;
4      private Address address;
5
6      public Person(String name, Address address) {
7          this.name = name;
8          this.address = address;
9      }
10
11     public String name() {
12         return this.name;
```

```
    }

    public void name(String name) {
        this.name = name;
    }

    public Address address() {
        return this.address;
    }

    public void address(Address address) {
        this.address = address;
    }

    @Override
    public Person clone() {
        try {
            return (Person) super.clone();
        } catch (CloneNotSupportedException e) {
            throw new AssertionError();
        }
    }
}
```

Address

```
public class Address implements Cloneable {

    private String city;
    private String street;

    public Address(String city, String street) {
        this.city = city;
        this.street = street;
    }

    public String city() {
        return this.city;
    }

    public void city(String city) {
        this.city = city;
    }

    public String street() {
        return this.street;
    }

    public void street(String street) {
        this.street = street;
    }

    @Override
    public Address clone() {
        try {
            return (Address) super.clone();
        } catch (CloneNotSupportedException e) {
```

```
            throw new AssertionError();
        }
    }
}
```

Client

```
class Client {

    public static void main(String[] args) {

        Person original = new Person("Original name", new Address("Original city", "Original street"));
        print("Original", original);

        // Clone as a shallow copy
        Person clone = original.clone();
        print("Clone", clone);

        clone.name("Modified name");
        clone.address().city("Modified city");

        print("Clone after modification", clone);
        print("Original after clone modification", original);
    }

    private static void print(String object, Person person) {
        System.out.println(String.format("%s: %s, %s",
                object,
                person.name(),
                person.address().city()));
    }
}
```

Output

```
Original: Original name, Original city
Clone: Original name, Original city
Clone after modification: Modified name, Modified city
Original after clone modification: Original name, Modified city
```

6.1.5.2 Example: `Person - Deep copy`

Address

```java
public class Address implements Cloneable {

    private String city;
    private String street;

    public Address(String city, String street) {
        this.city = city;
        this.street = street;
    }

    public String city() {
        return this.city;
    }

    public void city(String city) {
        this.city = city;
    }

    public String street() {
        return this.street;
    }

    public void street(String street) {
        this.street = street;
    }

    @Override
    public Address clone() {
        try {
            return (Address) super.clone();
        } catch (CloneNotSupportedException e) {
            throw new AssertionError();
        }
    }
}
```

Person

```java
public class Person implements Cloneable {

    private String name;
    private Address address;

    public Person(String name, Address address) {
        this.name = name;
        this.address = address;
    }

    public String name() {
        return this.name;
    }

    public void name(String name) {
```

```
        this.name = name;
    }

    public Address address() {
        return this.address;
    }

    public void address(Address address) {
        this.address = address;
    }

    @Override
    public Person clone() {
        try {
            Person person = (Person) super.clone();
            person.address(person.address.clone());

            return person;
        } catch (CloneNotSupportedException e) {
            throw new AssertionError();
        }
    }
}
```

Client

```
class Client {

    public static void main(String[] args) {

        Person original = new Person("Original name", new Address("Original city", "Original street"));
        print("Original", original);

        // Clone as a deep copy
        Person clone = original.clone();
        print("Clone", clone);

        clone.name("Modified name");
        clone.address().city("Modified city");

        print("Clone after modification", clone);
        print("Original after clone modification", original);
    }

    private static void print(String object, Person person) {
        System.out.printf("%s: %s, %s%n",
                object,
                person.name(),
                person.address().city());
    }
}
```

Output

```
Original: Original name, Original city
Clone: Original name, Original city
Clone after modification: Modified name, Modified city
Original after clone modification: Original name, Original city
```

6.1.5.3 Example: Person - Copy constructor / factory

According to Effective Java `clone` and `java.lang.Cloneable` is complex and the rules for an overriding clone are tricky and difficult to get right.

Object's `clone` method is very tricky. It's based on field copies, and it's *extra-linguistic*. It creates an object without calling a constructor. There are no guarantees that it preserves the invariants established by the constructors. If that state is mutable, you don't have two independent objects. If you modify one, the other changes as well. And suddenly, you get random behavior. A better approach is providing a *copy constructor* or *copy factory*. [3]

Person

```
public class Person {

    private String name;
    private Address address;

    // Copy constructor
    public Person(Person person) {
        this(person.name(), new Address(person.address()));
    }

    public Person(String name, Address address) {
        this.name = name;
        this.address = address;
    }

    public String name() {
        return this.name;
    }

    public void name(String name) {
        this.name = name;
    }

    public Address address() {
        return this.address;
    }

    public void address(Address address) {
        this.address = address;
    }

    // Copy factory
    public static Person newInstance(Person person) {
        return new Person(person.name(), new Address(person.address()));
```

[3] Bloch, Joshua: Effective Java – Third Edition, Item 13

```
    }
}
```

Address

```
public class Address {

    private String city;
    private String street;

    // Copy constructor
    public Address(Address address) {
        this(address.city(), address.street());
    }

    public Address(String city, String street) {
        this.city = city;
        this.street = street;
    }

    public String city() {
        return this.city;
    }

    public void city(String city) {
        this.city = city;
    }

    public String street() {
        return this.street;
    }

    public void street(String street) {
        this.street = street;
    }

    // Copy factory
    public static Address newInstance(Address address) {
        return new Address(address.city(), address.street());
    }
}
```

Client

```
class Client {

    public static void main(String[] args) {

        Person original = new Person("Original name", new Address("Original city", "Original street"));
        print("Original", original);

        // Clone as a deep copy with copy constructor
        Person clone = new Person(original);
        // Alternative as a deep copy with copy factory
        // Person clone = Person.newInstance(original);
        print("Clone", clone);
```

```
13
14          clone.name("Modified name");
15          clone.address().city("Modified city");
16
17          print("Clone after modification", clone);
18          print("Original after clone modification", original);
19      }
20
21      private static void print(String object, Person person) {
22          System.out.printf("%s: %s, %s%n",
23                  object,
24                  person.name(),
25                  person.address().city());
26      }
27  }
```

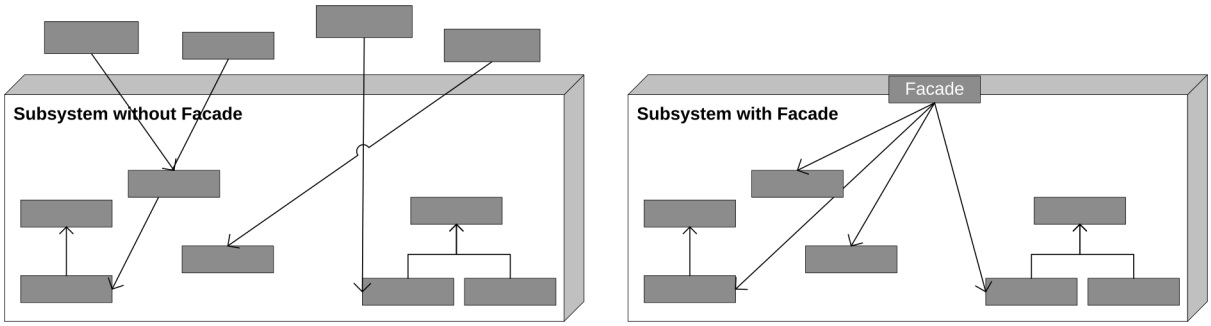
Output

```
1  Original: Original name, Original city
2  Clone: Original name, Original city
3  Clone after modification: Modified name, Modified city
4  Original after clone modification: Original name, Original city
```

6.2 Structural

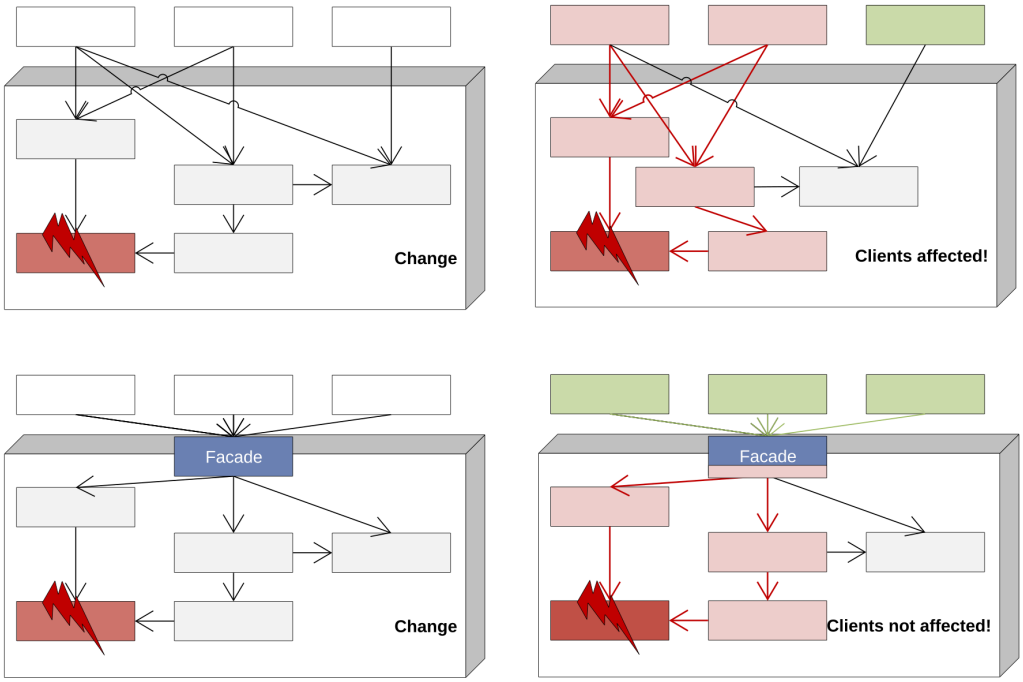
6.2.1 Facade

Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.



Use cases:

- Division of a complex system into several simple subsystems.
- Facade promotes loose coupling.
- Subdivision of a system into layers and the communication of these via facades.
- Reduction of dependencies between the client and the subsystem used.



6.2.1.1 Example: `Travel`

HotelBooker

```
public class HotelBooker {
    public List<Hotel> hotels(LocalDate from, LocalDate to) {
        // returns hotels available in the particular date range
    }
}
```

FlightBooker

```
public class FlightBooker {
    public List<Flight> flights(LocalDate from, LocalDate to) {
        // returns flights available in the particular date range
    }
}
```

TravelFacade

```
public class TravelFacade {
    private HotelBooker hotelBooker;
    private FlightBooker flightBooker;

    public Travels availableTravels(LocalDate from, LocalData to) {
        List<Flight> flights = flightBooker.flights(from, to);
        List<Hotel> hotels = hotelBooker.hotels(from, to);

        // process
     }
}
```

Client

```
class Client {
     public static void main(String[] args) {
         TravelFacade facade = new TravelFacade();
         facade.availableTravels(from, to);
     }
}
```

6.2.1.2 Example: `SmartHome`

LightController

```
class LightController {

  public void turnOnLights() {
    System.out.println("Turning on lights");
  }

  public void turnOffLights() {
    System.out.println("Turning off lights");
  }
}
```

SecuritySystem

```
class SecuritySystem {

  public void armSystem() {
    System.out.println("Arming security system");
  }

  public void disarmSystem() {
    System.out.println("Disarming security system");
  }
}
```

ThermostatController

```
class ThermostatController {

  public void setTemperature(int temperature) {
    System.out.println("Setting temperature to " + temperature + " degrees Celsius");
  }
}
```

SmartHomeFacade

```
class SmartHomeFacade {

  private final LightController lightController;
  private final ThermostatController thermostatController;
  private final SecuritySystem securitySystem;

  SmartHomeFacade() {
    this.lightController = new LightController();
    this.thermostatController = new ThermostatController();
    this.securitySystem = new SecuritySystem();
  }

  public void leaveHome() {
    this.lightController.turnOffLights();
    this.thermostatController.setTemperature(18);
    this.securitySystem.armSystem();
  }

```

```
19    public void returnHome() {
20      this.securitySystem.disarmSystem();
21      this.lightController.turnOnLights();
22      this.thermostatController.setTemperature(22);
23    }
24  }
```

Client

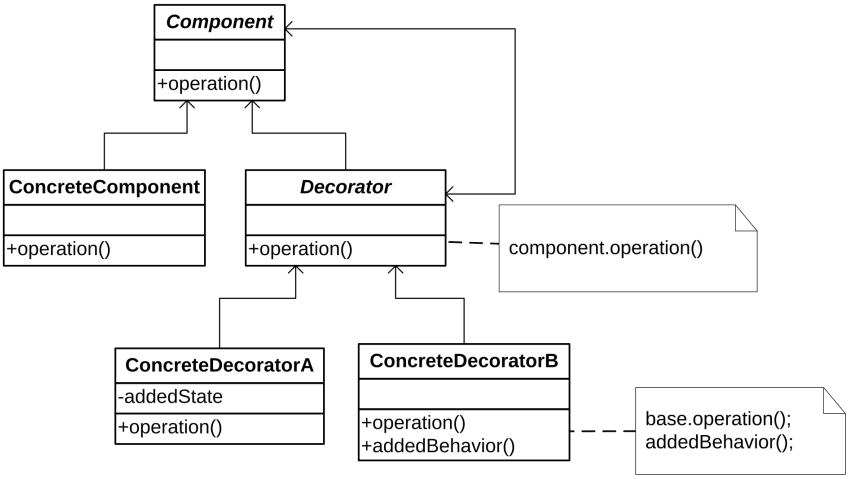
```
class Client {

  public static void main(String[] args) {
    SmartHomeFacade smartHomeFacade = new SmartHomeFacade();

    smartHomeFacade.leaveHome();
    smartHomeFacade.returnHome();
  }
}
```

Advantages:

- **Simplified interface**. The client can use a complex system more easily by reducing the number of objects that the client needs to know.
- **Decoupling** the client from the subsystem. Since the client only works against the facade, it is independent of changes in the subsystem.
- **Maintenance** and modifications of the subsystem only mean changes within the system and at most the facade implementation. The interface of the facade to the outside remains unchanged.

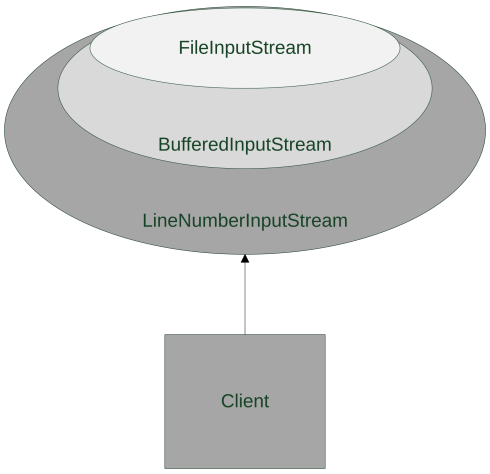
6.2.2 Decorator

Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.



Use cases:

- Alternative to static subclassing (inheritance) of objects
- Dynamic adding or removing functionalities
- Extension of `final` classes, which are therefore not inheritable
- If functionality extension using static inheritance is impracticable
  - z. e.g., with an unmanageable number of classes, which would result if every possible combination of extensions were considered
- Practical examples:
  - Java IO API, `BufferedInputStream` extends the interface around a buffer
  - Different types of cars with different features (lowering, spoiler)
  - Dish with various side dishes
  - Different types of a telephone (mobile phone, smartphone, classic telephone) with variable behavior (vibration, ringing, silent, lights)



6.2.2.1 Example: `Message`

Message

```
public interface Message {
    void print(String text);
}
```

DefaultMessage

```
public class DefaultMessage implements Message {

    @Override
    public void print(String text) {
        System.out.println(text);
    }
}
```

TimeStampMessage

```
public class TimeStampMessage implements Message {

    private final Message message;

    public TimeStampMessage(Message message) {
        this.message = message;
    }

    @Override
    public void print(String text) {
        this.message.print(String.format("%s %s", LocalDateTime.now(), text));
    }
}
```

UpperCaseMessage

```
public class UpperCaseMessage implements Message {

    private final Message message;

    public UpperCaseMessage(Message message) {
        this.message = message;
    }

    @Override
    public void print(String text) {
        this.message.print(text.toUpperCase());
    }
}
```

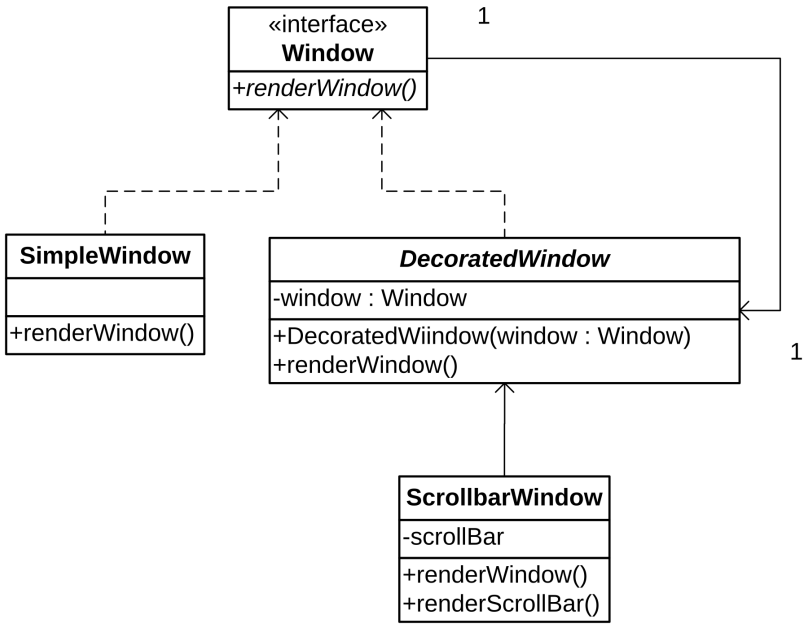
Client

```
class Client {

    public static void main(String[] args) {
        new DefaultMessage().print("DefaultMessage");
        new UpperCaseMessage(new DefaultMessage()).print("DefaultMessage");
        new TimeStampMessage(new DefaultMessage()).print("DefaultMessage");
        new UpperCaseMessage(new TimeStampMessage(new DefaultMessage())).print("DefaultMessage");
    }
}
```

Output

```
DefaultMessage
DEFAULTMESSAGE
2020-12-29T13:04:38.388325800 DefaultMessage
2020-12-29T13:04:38.416309600 DEFAULTMESSAGE
```

6.2.2.2 Example: `Window`



Window

```
public interface Window {
    void renderWindow();
}
```

SimpleWindow

```
public class SimpleWindow implements Window {
    public void renderWindow() {
        // implementation of rendering details
    }
}
```

DecoratedWindow

```
public class DecoratedWindow implements Window {
    private Window window = null;

    public DecoratedWindow(Window window) {
        this.window = window;
    }

    public void renderWindow() {
        window.renderWindow();
    }
}
```

ScrollableWindow

```
public class ScrollableWindow extends DecoratedWindow {
    private Object scrollBar = null;

    public ScrollableWindow(Window window) {
        super(window);
    }

    public void renderWindow() {
        super.renderWindow();
        renderScrollBar();
    }

    private void renderScrollBar() {
        scrollBar = new Object(); // prepare scrollbar
        // render scrollbar
    }
}
```

Client

```
class Client {
    public static void main(String[] args) {
        Window window = new SimpleWindow();
        window.renderWindow();

        // at some point later maybe text size becomes larger
        // than the window thus the scrolling behavior must be added

        // decorate old window, now window object has additional behavior
        window = new ScrollableWindow(window);

```

```
12          window.renderWindow();
13      }
14  }
```

Advantages:

- Decorator offers **more possibilities** than inheritance, as it is not static and therefore defines a behavior of the subclass.
- **Slim**, **cohesive classes**. Each decorator represents exactly one function and nothing else. This increases class cohesion and makes the corresponding code easier to maintain and extend.
- Avoidance of long and confusing inheritance hierarchies.
- By combining a few Decorator objects, the functionality of the object can be extended considerably.

Disadvantages:

- Decorated objects are wrapped by the actual decorator. This makes it difficult to determine the identity of the objects.
- **High number** of objects. Each additional feature requires a new Decorator object. The number of many small, similar objects and their initialization code can quickly become confusing. Can be fixed by a factory.
- Decorator-Pattern is a **dynamic process**, so in general the runtime behavior is worse than with classical inheritance.

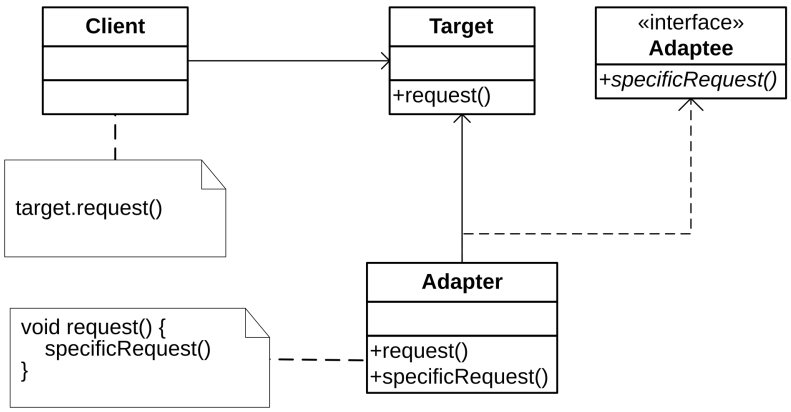
6.2.3 Adapter

Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

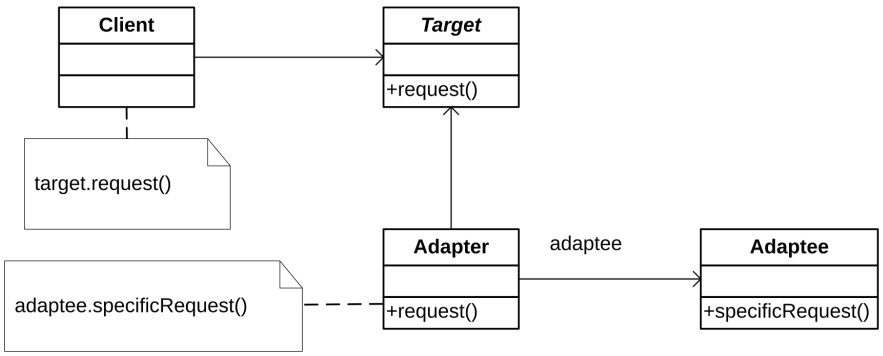
The adapter pattern defines four roles:

- **Adaptee** which is to be adapted to the target interface.
- **Adapter** who performs the adaptation.
- **Target interface** to be used by the client.
- **Client** who uses the adaptee through the adapter.

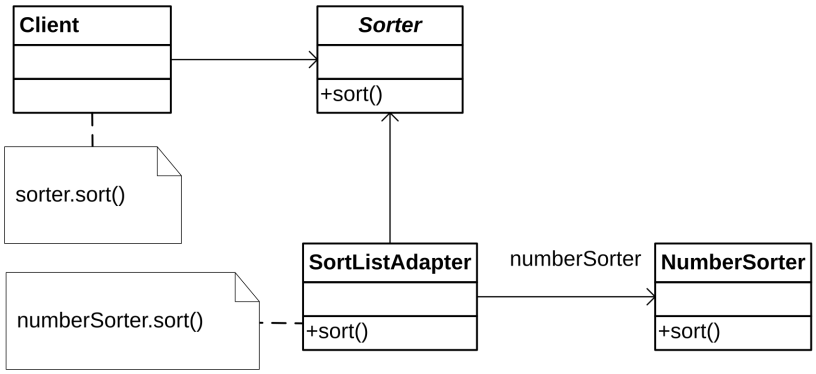
Class Adapter uses multiple inheritance:
In the case of the class adapter, the adapter inherits our adaptee class and implements the target interface. When a client object calls a method of the class adapter, the adapter internally calls a method of the adapter it inherits.



Object Adapter relies on object composition:
In the case of an object adapter, the adapter delegates to an instance of the adaptee. The adapter implements the target interface but does not inherit the classes that need to be adapted. When a client calls a method of the object adapter, the object adapter calls a corresponding method on the instance of the adaptee it references.



6.2.3.1 Example: `Sorter`



Sorter - Target interface

```
public interface Sorter {
    int[] sort(int[] numbers);
}
```

SortListAdapter - Adapter

```
public class SortListAdapter implements Sorter {

    public int[] sort(int[] numbers) {
        List<Integer> numberList = convertArrayToList(numbers);

        NumberSorter numberSorter = new NumberSorter();
        numberList = numberSorter.sort(numberList);

        return convertListToArray(numberList);
    }
}
```

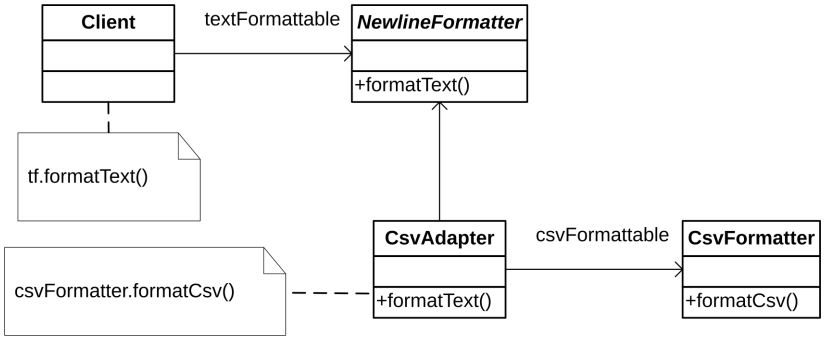
NumberSorter - Adaptee

```
// A third party implementation of a number sorter that deals with lists, not arrays.
public class NumberSorter {
    public List<Integer> sort(List<Integer> numbers) {
        return sort(numbers);
    }
}
```

Client

```
class Client {
    public static void main(String[] args) {
        Sorter sorter = new SortListAdapter();
        sorter.sort(new int[] { 34, 2, 4, 12, 1 });
    }
}
```

6.2.3.2 Example: `TextFormatter`



TextFormattable

```
public interface TextFormattable {
    String formatText(String text, String separator);
}
```

NewlineFormatter

```
public class NewlineFormatter implements TextFormattable {

    @Override
    public String formatText(String text, String separator) {
        if (text != null && separator != null) {
            return Stream.of(text.split(Pattern.quote(separator)))
                    .map(String::stripLeading)
                    .collect(Collectors.joining(System.lineSeparator()));
        }

        return null;
    }
}
```

CsvFormattable

```
public interface CsvFormattable {
    String formatCsv(String text, String separator);
}
```

CsvFormatter

```
public class CsvFormatter implements CsvFormattable {

    @Override
    public String formatCsv(String text, String separator) {
        if (text != null && separator != null) {
            return Stream.of(text.split(Pattern.quote(separator)))
                    .collect(Collectors.joining(", "));
        }

        return null;
    }
}
```

CsvAdapter

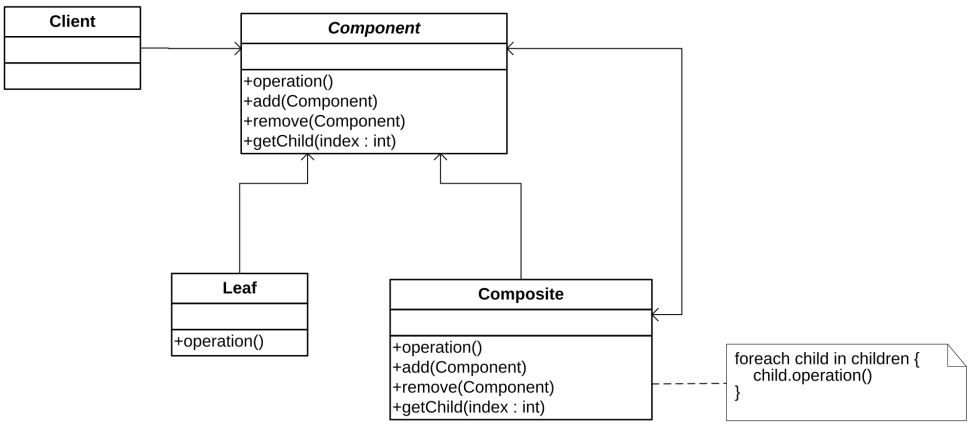
```
public class CsvAdapter implements TextFormattable {

    private final CsvFormatter csvFormatter;

    public CsvAdapter(CsvFormatter csvFormatter) {
        this.csvFormatter = csvFormatter;
    }

    @Override
    public String formatText(String text, String separator) {
        return this.csvFormatter.formatCsv(text, separator);
    }
}
```

Client

```
class Client {

    public static void main(String[] args) {
        String text = "Adapter Pattern 1. Adapter Pattern 2. Adapter Pattern 3.";

        TextFormattable tf = new NewlineFormatter();
        System.out.println(tf.formatText(text, "."));

        tf = new CsvAdapter(new CsvFormatter());
        System.out.println(tf.formatText(text, "."));
    }
}
```

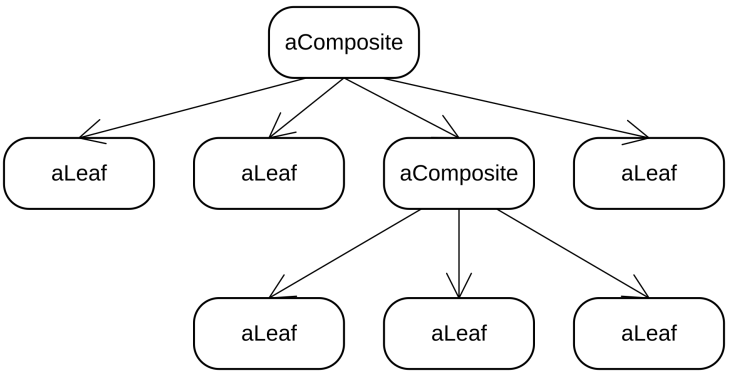
6.2.4 Composite

Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.



Use cases:

- Manipulation of a hierarchical collection of simple and complex objects.
- Modelling of a tree-like structure of objects
  - **File system**. Files and folders can be modeled with the composite pattern. Files represent the leafs and folders represent the composites, as they may contain other files and folders.
  - **Menus**. Menus consist of a root entry (composite) and commands (leafs).
  - Hierarchy of graphic objects e.g., in Java / AWT (Container & Component)



6.2.4.1 Example: `Graphic`

Component

```
public interface Component {
    void paint();
}
```

Composite

```
public abstract class Composite implements Component {

    private List<Component> components = new ArrayList<Component>();

    @Override
    public void paint() {
        for (Component component : this.components) {
            component.paint();
        }
    }

    public void add(Component component) {
        this.components.add(component);
    }

    public void remove(Component component) {
        this.components.remove(component);
    }

    public Component get(int index) {
        return this.components.get(index);
    }
}
```

Sheet

```
public class Sheet extends Composite {
    public void paint() {
        System.out.println(Sheet.class.getSimpleName());
        super.paint();
    }
}
```

Row

```
public class Row extends Composite {
    public void paint() {
        System.out.printf(" %s%n", Row.class.getSimpleName());
        super.paint();
    }
}
```

Column

```
public class Column extends Composite {
    public void paint() {
        System.out.printf("  %s%n", Column.class.getSimpleName());
        super.paint();
    }
}
```

Client

```
class Client {
    public static void main(String[] args) {
        Composite sheet = new Sheet();

        Composite r1 = new Row();
        r1.add(new Column());
        r1.add(new Column());
        sheet.add(r1);

        Composite r2 = new Row();
        r2.add(new Column());
        r2.add(new Column());
        sheet.add(r2);

        sheet.paint();
    }
}
```

Output

```
Sheet
 Row
  Column
  Column
 Row
  Column
  Column
```

6.2.4.2 Example: `Organization Chart`

Employee

```
public interface Employee {
    void draw();
}
```

DisciplinaryLeadership

```
public abstract class DisciplinaryLeadership implements Employee {

    private final List<Employee> employees = new ArrayList<>();

    @Override
    public void draw() {
        for (Employee employee : this.employees) {
            employee.draw();
        }
    }

    public void add(Employee employee) {
        this.employees.add(employee);
    }

    public Employee get(int index) {
        return this.employees.get(index);
    }
}
```

CTO

```
public class CTO extends DisciplinaryLeadership {

    @Override
    public void draw() {
        System.out.println(CTO.class.getSimpleName());
        super.draw();
    }
}
```

VP

```
public class VP extends DisciplinaryLeadership {

    @Override
    public void draw() {
        System.out.printf(" %s%n", VP.class.getSimpleName());
        super.draw();
    }
}
```

Developer

```java
public class Developer implements Employee {

    @Override
    public void draw() {
        System.out.printf("  %s%n", Developer.class.getSimpleName());
    }
}
```

Client

```java
class Client {
    public static void main(String[] args) {

        DisciplinaryLeadership cto = new CTO();
        DisciplinaryLeadership vp1 = new VP();
        DisciplinaryLeadership vp2 = new VP();

        vp1.add(new Developer());
        vp1.add(new Developer());

        vp2.add(new Developer());
        vp2.add(new Developer());
        vp2.add(new Developer());

        cto.add(vp1);
        cto.add(vp2);

        cto.draw();
    }
}
```

Output

```
CTO
 VP
  Developer
  Developer
 VP
  Developer
  Developer
  Developer
```

Advantages:

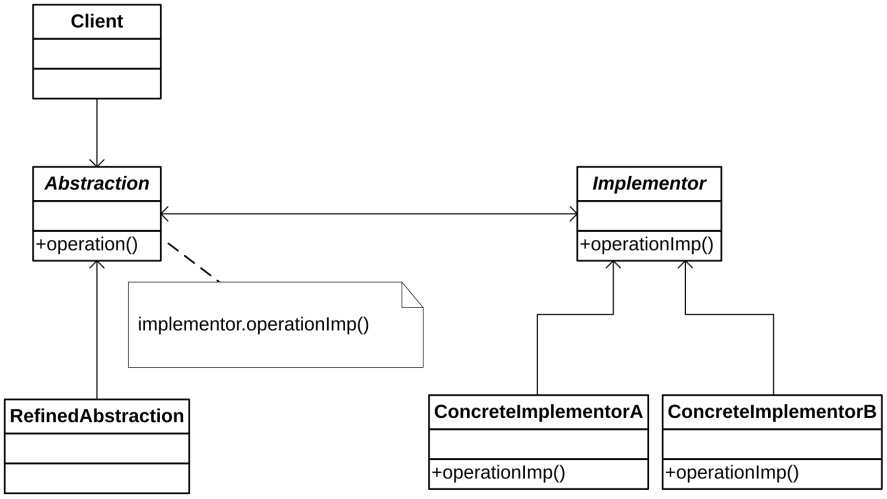
- Simple implementation of **nested** structures
- **Elegant working** with the tree structure. The client can easily traverse through the hierarchy, call operations, and manage them, i.e., add new elements and delete existing ones.
- **Flexibility and expandability**. Easy addition of new elements (leafs or composites).

Disadvantages:

- **Generalisation of the draft**. If a particular compound element is to have only a fixed number of children or only certain children, this can only be checked at runtime.

6.2.5 Bridge

Decouple an abstraction from its implementation so that the two can vary independently.



Use cases:

- Avoiding a permanent binding between an abstraction and its implementation.
- Combining different abstractions and implementations and extending them independently.
- Changes in the implementation of an abstraction should have no impact on clients.
- Sharing an implementation among multiple objects.
- Run-time binding of the implementation.

6.2.5.1 Example: `Message`

Message

```
public abstract class Message {

    private final MessageSender messageSender;

    protected Message(MessageSender messageSender) {
        this.messageSender = messageSender;
    }

    public MessageSender messageSender() {
        return this.messageSender;
    }

    abstract void send();
}
```

EmailMessage

```
public class EmailMessage extends Message {

    public EmailMessage(MessageSender messageSender) {
        super(messageSender);
    }

    @Override
    public void send() {
        messageSender().sendMessage();
    }
}
```

SmsMessage

```
public class SmsMessage extends Message {

    public SmsMessage(MessageSender messageSender) {
        super(messageSender);
    }

    @Override
    public void send() {
        messageSender().sendMessage();
    }
}
```

MessageSender

```
public interface MessageSender {

    void sendMessage();
}
```

EmailMessageSender

```
public class EmailMessageSender implements MessageSender {

    @Override
    public void sendMessage() {
        System.out.println("Sending email message...");
    }
}
```

SmsMessageSender

```
public class SmsMessageSender implements MessageSender {

    @Override
    public void sendMessage() {
        System.out.println("Sending sms message...");
    }
}
```

Client

```
class Client {

    public static void main(String[] args) {
        sendSms();
        sendEmail();
    }

    private static void sendSms() {
        MessageSender sender = new SmsMessageSender();
        Message message = new SmsMessage(sender);
        message.send();
    }

    private static void sendEmail() {
        MessageSender sender = new EmailMessageSender();
        Message message = new EmailMessage(sender);
        message.send();
    }
}
```

6.2.5.2 Example: `Television`

Font

```
public interface Television {

    void on();

    void off();

    void channel(int channel);
}
```

Samsung

```
public class Samsung implements Television {

    @Override
    public void on() {
        System.out.println("On");
    }

    @Override
    public void off() {
        System.out.println("Off");
    }

    @Override
    public void channel(int channel) {
        System.out.println(channel);
    }
}
```

Philips

```
public class Philips implements Television {

    @Override
    public void on() {
        System.out.println("On");
    }

    @Override
    public void off() {
        System.out.println("Off");
    }

    @Override
    public void channel(int channel) {
        System.out.println(channel);
    }
}
```

RemoteControl

```
public abstract class RemoteControl {

    private final Television television;

    protected RemoteControl(Television television) {
        this.television = television;
    }

    public void on() {
        this.television.on();
    }

    public void off() {
        this.television.off();
    }

    public void channel(int channel) {
        this.television.channel(channel);
    }
}
```

SmartRemoteControl

```
public class SmartRemoteControl extends RemoteControl {

    private int channel;

    public SmartRemoteControl(Television television) {
        super(television);
    }

    public void nextChannel() {
        this.channel++;
        channel(this.channel);
    }

    public void previousChannel() {
        this.channel--;
        channel(this.channel);
    }
}
```

Client

```
class Client {

    public static void main(String[] args) {
        useSamsung();
        usePhilips();
    }

    private static void useSamsung() {
        Television television = new Samsung();
        SmartRemoteControl remoteControl = new SmartRemoteControl(television);
```

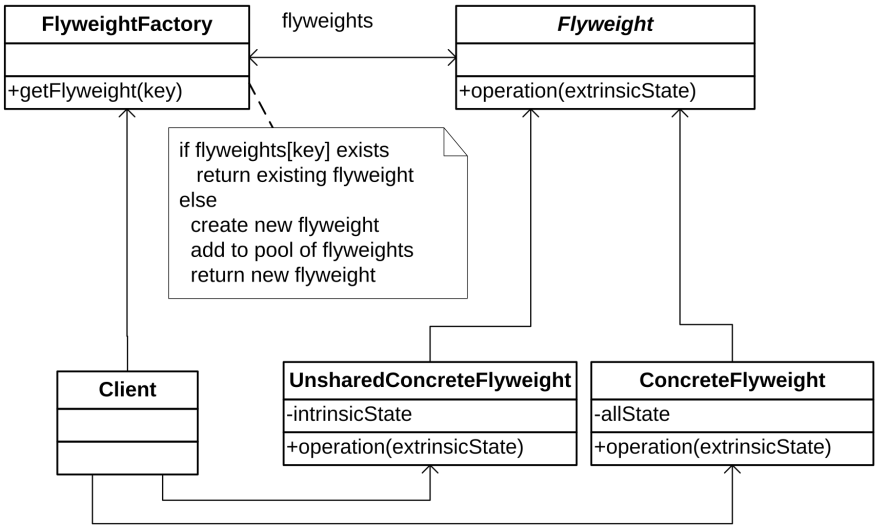
```
11
12          remoteControl.on();
13          remoteControl.nextChannel();
14          remoteControl.nextChannel();
15          remoteControl.previousChannel();
16          remoteControl.off();
17      }
18
19      private static void usePhilips() {
20          Television television = new Philips();
21          SmartRemoteControl remoteControl = new SmartRemoteControl(television);
22
23          remoteControl.on();
24          remoteControl.nextChannel();
25          remoteControl.nextChannel();
26          remoteControl.previousChannel();
27          remoteControl.off();
28      }
29  }
```

Advantages:

- Abstraction and implementation are **decoupled** and **independent**.
- Improved **extensibility**, you can extend the abstraction and implementation hierarchies independently.
- The implementation is also changed **dynamically at run time**.
- **Hiding** of implementation details from the client.

6.2.6 Flyweight

Use sharing to support large numbers of fine-grained objects efficiently.



Use cases:

- Need to create **large** number of objects when memory cost is a constraint.
- When many of the object attributes can be made **external and shared**.
- Avoid unnecessary object **initialization**.
- Where the **identity** of the object is not a matter of concern and the required operations can be performed on the shared objects.

As defined by GoF, an object can have two states, the intrinsic and the extrinsic state:

Intrinsic state
: Stored in the flyweight; it consists of information that's independent of the flyweights context, thereby making it shareable.

Extrinsic state
: Depends on and varies with the flyweight's context and therefore cannot be shared. Client objects are responsible for passing extrinsic state to the flyweight if necessary.

6.2.6.1 Example: `Font`

Font

```
public record Font(String name, int size) {

}
```

FontFactory

```
public final class FontFactory {

    private final Set<Font> fonts = new HashSet<>();

    public Font of(String name, int size) {
        for (Font font : this.fonts) {
            if (font.name().equals(name) && font.size() == size) {
                return font;
            }
        }

        Font font = new Font(name, size);
        this.fonts.add(font);

        return font;
    }
}
```

Client

```
class Client {

    public static void main(String[] args) {
        FontFactory factory = new FontFactory();

        System.out.println(factory.of("Helvetica", 12));
        System.out.println(factory.of("Arial ", 10));

        // Will return same objects
        System.out.println(factory.of("Helvetica", 12));
        System.out.println(factory.of("Arial ", 10));
    }
}
```

6.2.6.2 Example: `City`

Person

```
public record Person(String firstName, String lastName, String street, City city) {

}
```

City

```
public record City(String country, String countryCode, String name) {

}
```

CityFactory

```
public final class CityFactory {

    private final Set<City> cities = new HashSet<>();

    public City of(String country, String countryCode, String name) {
        for (City city : this.cities) {
            if (city.country().equals(country)
                    && city.countryCode().equals(countryCode)
                    && city.name().equals(name)) {
                return city;
            }
        }

        City city = new City(country, countryCode, name);
        this.cities.add(city);

        return city;
    }
}
```

Employee

```
class Client {

    public static void main(String[] args) {
        CityFactory factory = new CityFactory();

        // Will use same city instance
        new Person("Forster", "Gamlen", "7654 Ruskin Center",
            factory.of("United States", "US", "Carson City"));
        new Person("Nevin", "Roddell", "5 Del Sol Alley",
            factory.of("United States", "US", "Carson City"));
        new Person("Hunter", "Lewins", "33733 Heffernan Circle",
            factory.of("United States", "US", "Carson City"));

        new Person("Reube", "Gregoretti", "08 Melrose Street",
            factory.of("Japan", "JP", "Mutsu"));
    }
}
```

Advantages:

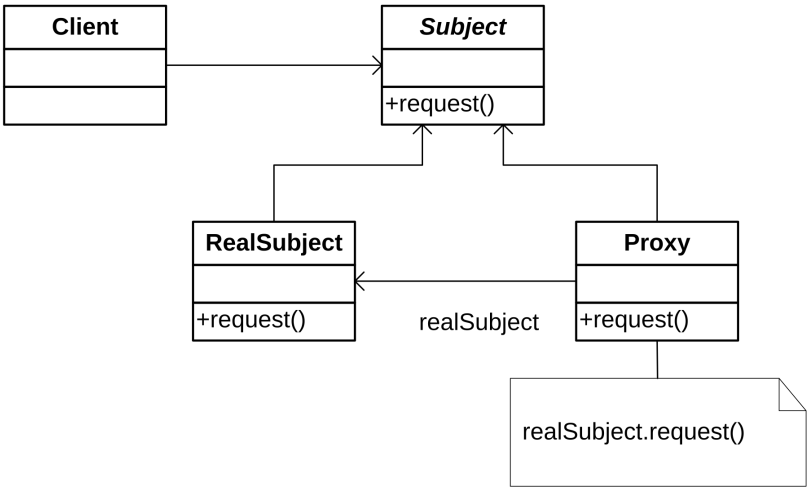
- Improves the performance by **reducing** the number of objects.
- **Reduces the memory** footprint as the common properties are shared between objects using intrinsic attributes.

Disadvantages:

- If there are **no shareable attributes** in an object, the pattern is of no use.
- May introduce **run-time costs** associated with transferring, finding, and/or computing extrinsic state, especially if it was formerly stored as an intrinsic state.

6.2.7 Proxy

Provide a surrogate or placeholder for another object to control access to it.



Use cases:

- Proxies are often used in situations where a more complex reference to an object is required instead of a normal pointer. A distinction is usually made between the following types of proxies:
  - **Remote proxy** provides a local representation of another remote object or resource.
  - **Virtual proxy** wrap expensive objects and loads them on-demand.
  - **Protection proxy** provides access control to an original object.
  - **Smart reference** executes additional operations when an object is accessed.

6.2.7.1 Example: `Spaceship`

Spaceship

```
interface Spaceship {
    void fly();
}
```

Pilot

```
public record Pilot(String name) {

}
```

MillenniumFalcon

```
public final class MillenniumFalcon implements Spaceship {

    @Override
    public void fly() {
        System.out.println("Welcome, Han!");
    }
}
```

MillenniumFalconProxy

```
public final class MillenniumFalconProxy implements Spaceship {

    private final Pilot pilot;
    private final Spaceship ship;

    public MillenniumFalconProxy(Pilot pilot) {
        this.pilot = pilot;
        this.ship = new MillenniumFalcon();
    }

    @Override
    public void fly() {
        if ("Han Solo".equals(this.pilot.name())) {
            this.ship.fly();
        } else {
            System.out.printf("Sorry %s, only Han Solo can fly the Falcon!%n", this.pilot.name());
        }
    }
}
```

Client

```
class Client {

    public static void main(String[] args) {
        Spaceship ship = new MillenniumFalconProxy(new Pilot("Han Solo"));
        ship.fly();

        ship = new MillenniumFalconProxy(new Pilot("Darth Vader"));
        ship.fly();
    }
}
```

6.2.7.2 Example: `ImageViewer`

ImageViewer

```
interface ImageViewer {
    void display();
}
```

DefaultImageViewer

```
public final class DefaultImageViewer implements ImageViewer {

    private final Image image;

    public DefaultImageViewer(String path) {
        // Slow operation
        this.image = Image.load(path);
    }

    @Override
    public void display() {
        // Slow operation
        this.image.display();
    }
}
```

Image

```
public interface Image {

    void display();

    static Image load(String path) {
        // Not implemented
        return null;
    }
}
```

ImageViewerProxy

```
public final class ImageViewerProxy implements ImageViewer {

    private final String path;
    private ImageViewer viewer;

    public ImageViewerProxy(String path) {
        this.path = path;
    }

    @Override
    public void display() {
        if (this.viewer == null) {
            this.viewer = new DefaultImageViewer(this.path);
        }

        this.viewer.display();
    }
}
```

Client

```
class Client {

    public static void main(String[] args) {
        ImageViewer beer = new ImageViewerProxy("./img/beer.png");
        ImageViewer hammock = new ImageViewerProxy("./img/hammock.png");

        beer.display();
        beer.display(); // DefaultImageViewer will be call and be executed once

        hammock.display();
    }
}
```

Advantages:

- **Security**: Certain conditions can be checked when accessing the object.
- **Performance**: Objects with expensive operations demanding in terms of memory and execution time, can be wrapped so that they are called only when they are needed, or unnecessary instantiation can be avoided.

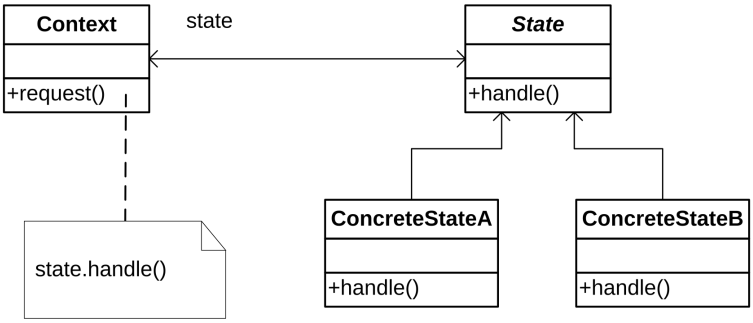
Disadvantages:

- **Performance**: This can also be a disadvantage of the proxy pattern if a proxy object is used to wrap an object that exists somewhere on the network. Since it is a proxy, it can hide from the client the fact that it is a remote communication.
- Add another layer of abstraction, which can lead to detours and increase complexity.

6.3 Behavioural

6.3.1 State

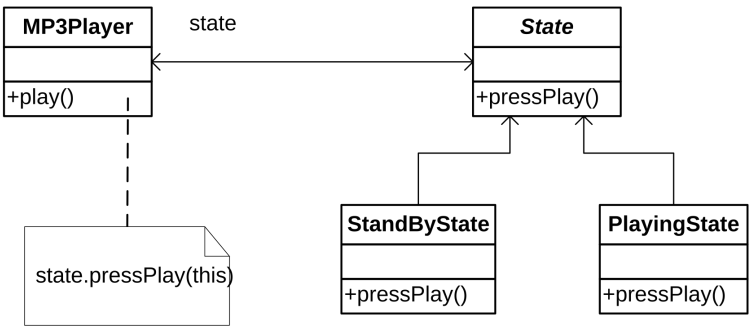
Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.



Use cases:

- An object should change its external behavior at runtime depending on its state.
  - **State machines** e.g., parser. The functionality of many text parsers is inevitably state-based. So, a compiler parsing source code must interpret a character depending on the characters read before.
  - Representation of states of network connections

6.3.1.1 Example: `MP3Player`



MP3Player

```
public class MP3Player {
    private State state;

    private MP3Player(State state) {
        this.state = state;
    }
    public void play() {
        state.pressPlay(this);
    }
    public void state(State state) {
        this.state = state;
    }
    public State state() {
        return state;
    }
}
```

State

```
public interface State {
    void pressPlay(MP3Player player);
}
```

PlayingState

```
public class PlayingState implements State {
    public void pressPlay(MP3Player player) {
        player.state(new StandByState());
    }
}
```

StandByState

```
public class StandByState implements State {
    public void pressPlay(MP3Player player) {
        player.state(new PlayingState());
    }
}
```

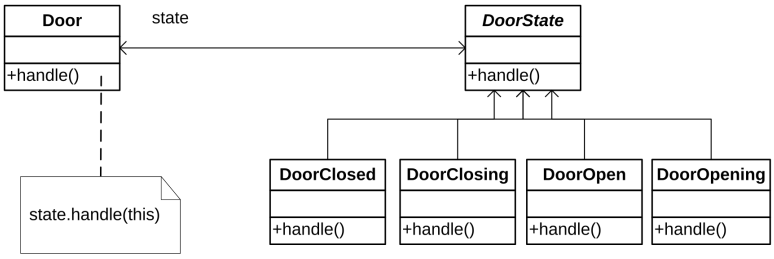
Client

```
class Client {
    public static void main(String[] args) {
        MP3Player player = new MP3Player(new PlayingState());
        player.play();
        player.play();
        player.play();
        player.play();
        player.play();
        player.play();
    }
}
```

Output

```
Play state
Standby state
Play state
Standby state
Play state
Standby state
Play state
```

6.3.1.2 Example: `Door`



Door

```
public class Door {

    private DoorState state;

    public Door(DoorState state) {
        this.state = state;
    }

    public void handle() {
        this.state.handle(this);
    }

    public void state(DoorState state) {
        this.state = state;
    }

    public DoorState state() {
        return this.state;
    }
}
```

DoorState

```
public interface DoorState {
    void handle(Door door);
}
```

DoorClosed

```
public class DoorClosed implements DoorState {

    public DoorClosed() {
        System.out.println("Door closed");
    }

    @Override
    public void handle(Door door) {
        door.state(new DoorOpening());
    }
}
```

DoorClosing

```
public class DoorClosing implements DoorState {

    public DoorClosing() {
        System.out.println("Door closing");
    }

    @Override
    public void handle(Door door) {
        door.state(new DoorClosed());
    }
}
```

DoorOpen

```
public class DoorOpen implements DoorState {

    public DoorOpen() {
        System.out.println("Door open");
    }

    @Override
    public void handle(Door door) {
        door.state(new DoorClosing());
    }
}
```

DoorOpening

```
public class DoorOpening implements DoorState {

    public DoorOpening() {
        System.out.println("Door opening");
    }

    @Override
    public void handle(Door door) {
        door.state(new DoorOpen());
    }
}
```

Client

```
class Client {
    public static void main(String[] args) {
        Door door = new Door(new DoorOpen());
        door.handle();
        door.handle();
        door.handle();
        door.handle();
        door.handle();
        door.handle();
    }
}
```

Output

```
Door open
Door closing
Door closed
Door opening
Door open
Door closing
Door closed
```

Advantages:

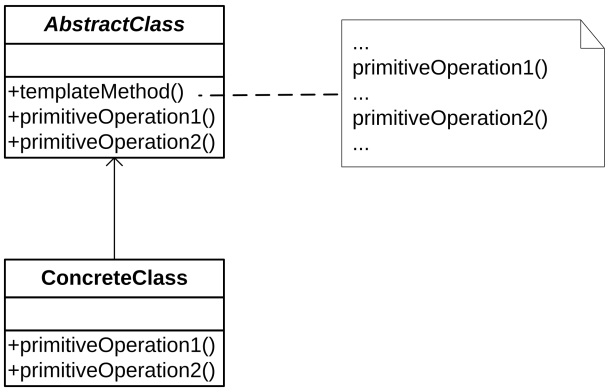
- **Extensibility and change stability**. Easy integration of new states without having to change existing code. Changes to state-dependent behavior only affect a state class and not the context.
- **Comprehensibility**. High cohesion and delegation of responsibilities (the behavior of a state is encapsulated/localized in the corresponding state object itself) make the code easy to understand.
- **Explicit state transitions**. By introducing objects for each state, the state transition becomes explicit. In addition, inconsistent states are prevented, because a state change is an atomic command (setting a variable).
- The design is **less error-prone** concerning later changes than broad, repetitive if-else constructs.

Disadvantages:

- **Increased number of classes**. Is less compact than a single class. However, the aim of the design pattern is precisely to distribute the behavior of a single confusing class to several state objects.

6.3.2 Template Method

Define a skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.



Use cases:

- Define a template that implements similar algorithms and delegates the individual execution steps to subclasses.
- Common behavior of subclasses should be moved to a superclass to avoid code duplication.
- Modelling of fixed sequences and the specification of variation points.

6.3.2.1 Example: `Compiler`

CrossCompiler

```
1  public abstract class CrossCompiler {
2      public final void crossCompile() {
3          collectSource();
4          compileToTarget();
5      }
6
7      // Template methods
8      protected abstract void collectSource();
9      protected abstract void compileToTarget();
10 }
```

IPhoneCompiler

```
public class IPhoneCompiler extends CrossCompiler {
    protected void collectSource() {
        // anything specific to this class
    }

    protected void compileToTarget() {
        // iphone specific compilation
    }
}
```

AndroidCompiler

```
public class AndroidCompiler extends CrossCompiler {
    protected void collectSource() {
        // anything specific to this class
    }

    protected void compileToTarget() {
        // android specific compilation
    }
}
```

Client

```
class Client {
    public static void main(String[] args) {
        CrossCompiler iphone = new IPhoneCompiler();
        iphone.crossCompile();

        CrossCompiler android = new AndroidCompiler();
        android.crossCompile();
    }
}
```

6.3.2.2 Example: `Callbackable`

Callbackable

```
interface Callbackable extends Runnable {
    void beforeRun();
    void afterRun();
    void onError(Exception e);
    void onSuccess();
}
```

AbstractCallbackRunnable

```
abstract class AbstractCallbackRunnable implements Callbackable {

    private final LocalDateTime submittedTime;

    private Status status;
    private LocalDateTime startedTime;
    private LocalDateTime finishedTime;

    AbstractCallbackRunnable() {
        this.submittedTime = LocalDateTime.now();
        this.status = Status.WAITING;
    }

    @Override
    public void run() {
        try {
            beforeRunIntern();
            startRun();
            onSuccessIntern();
        } catch (Exception e) {
            onErrorIntern(e);
        } finally {
            afterRunIntern();
        }
    }

    public abstract void startRun() throws Exception;

    private void beforeRunIntern() {
        this.startedTime = LocalDateTime.now();
        this.status = Status.RUNNING;

        beforeRun();
    }

    private void afterRunIntern() {
        this.finishedTime = LocalDateTime.now();
        afterRun();
    }

    private void onErrorIntern(Exception e) {
        this.status = Status.FAILED;
        onError(e);
    }

    private void onSuccessIntern() {
        this.status = Status.COMPLETED;
        onSuccess();
    }

    public Status status() {
        return this.status;
    }

    public LocalDateTime submittedTime() {
        return this.submittedTime;
```

```
57      }
58
59      public LocalDateTime startedTime() {
60          return this.startedTime;
61      }
62
63      public LocalDateTime finishedTime() {
64          return this.finishedTime;
65      }
66  }
```

Status

```
1   public enum Status {
2       UNKNOWN, WAITING, RUNNING, COMPLETED, FAILED
3   }
```

DefaultCallbackRunnable

```
1   public class DefaultCallbackRunnable extends AbstractCallbackRunnable {
2
3       @Override
4       public void beforeRun() {
5           System.out.println("beforeRun()");
6       }
7
8       @Override
9       public void afterRun() {
10          System.out.println("afterRun()");
11      }
12
13      @Override
14      public void onError(Exception e) {
15          System.out.println("onError()");
16      }
17
18      @Override
19      public void onSuccess() {
20          System.out.println("onSuccess()");
21      }
22
23      @Override
24      public void startRun() throws Exception {
25          System.out.println("startRun()");
26      }
27  }
```

Client

```
class Client {

    public static void main(String[] args) {
        ExecutorService executor = Executors.newFixedThreadPool(1);
        executor.submit(new DefaultCallbackRunnable());
    }
}
```

Advantages:

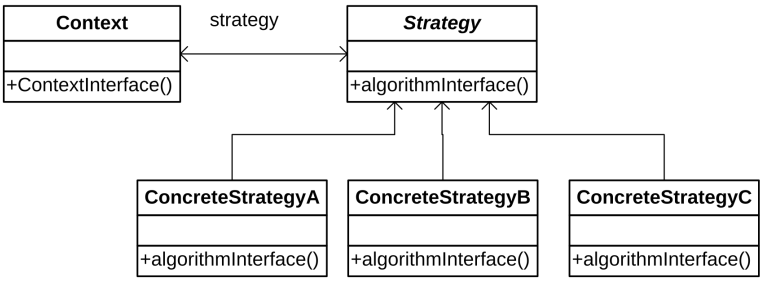
- The use of a template method allows inheriting classes to overwrite certain steps of an algorithm **without changing the structure of the algorithm**.
- Common behaviour is **encapsulated** in one class.
- **All methods** must be overwritten when implementing the subclass.
- Avoiding **duplicate** code.

Disadvantages:

- The design can become **unnecessarily complicated** if subclasses have to implement a large number of methods to make the algorithm more concrete.
- It can become a disadvantage if the algorithm follows a **fixed structure** that cannot be changed.
- It must be **clear** which methods may or must be overwritten.

6.3.3 Strategy

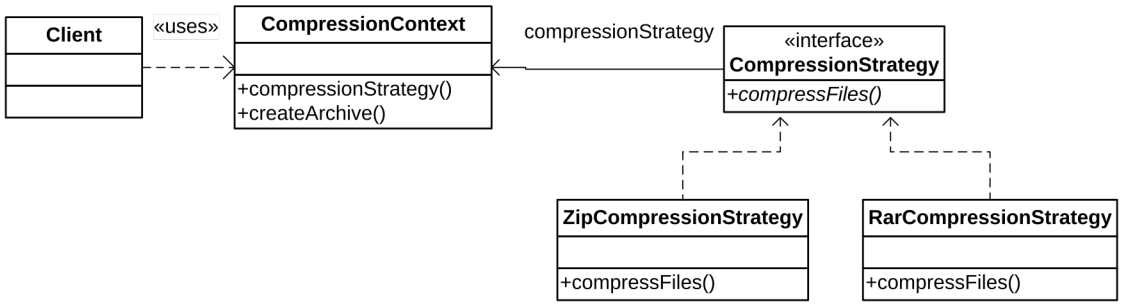
Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.



Use cases:

- For many similar classes that differ only in behavior.
- For input masks, the logic for validation and plausibility checks of user inputs can be encapsulated.
- Behaviour should also be decoupled from the context if different forms of the same function are required
  - Sorting of a collection (`Array`, `List`), whereby the concrete strategies represent different sorting methods.
  - Storage in different file formats.
  - Packer with different compression algorithms.
- Decoupling and hiding complicated algorithm details from the context.
- Multiple branches (`if ()... else if()... else ...`) can be avoided and this increases the clarity of the code.

6.3.3.1 Example: `Compression`



CompressionStrategy

```
public interface CompressionStrategy {
    void compressFiles(List<File> files);
}
```

RarCompressionStrategy

```
public class RarCompressionStrategy implements CompressionStrategy {
    public void compressFiles(List<File> files) {
        // using RAR approach
    }
}
```

ZipCompressionStrategy

```
public class ZipCompressionStrategy implements CompressionStrategy {
    public void compressFiles(List<File> files) {
        // using ZIP approach
    }
}
```

CompressionContext

```
public class CompressionContext {
    private CompressionStrategy strategy;

    // this can be set at runtime by the application preferences
    public void compressionStrategy(CompressionStrategy strategy) {
        this.strategy = strategy;
    }

    // use the strategy
    public void createArchive(List<File> files) {
        this.strategy.compressFiles(files);
    }
}
```

Client

```
class Client {

    public static void main(String[] args) {
        CompressionContext ctx = new CompressionContext();

        // we could assume context is already set by preferences
        ctx.compressionStrategy(new RarCompressionStrategy());

        // get a list of files

        ctx.createArchive(files);
    }
}
```

6.3.3.2 Example: `LogFormatter`

LogFormatter

```
interface LogFormatter {
    String format(Log log);
}
```

FullLogFormatter

```
public class FullLogFormatter implements LogFormatter {

    @Override
    public String format(Log log) {
        return String.format("Hash: %s%nAuthor: %s%nMessage: %s%nDate: %s%n",
                log.hash(),
                log.author(),
                log.message(),
                log.date());
    }
}
```

ShortLogFormatter

```
public class ShortLogFormatter implements LogFormatter {

    @Override
    public String format(Log log) {
        return String.format("Hash: %s%nAuthor: %s%nMessage: %s%n",
                log.hash(),
                log.author(),
                log.message());
    }
}
```

OnelineLogFormatter

```
public class OnelineLogFormatter implements LogFormatter {

    @Override
    public String format(Log log) {
        return String.format("%s %s",
                log.hash(),
                log.message());
    }
}
```

Log

```
public record Log(String hash, String author, String message, LocalDateTime date) {

}
```

GitConsole

```
public class GitConsole {

    private final LogFormatter formatter;

    public GitConsole(LogFormatter formatter) {
        this.formatter = formatter;
    }

    public void printLog(List<Log> logs) {
        logs.forEach(log -> System.out.println(this.formatter.format(log)));
    }
}
```

Client

```
class Client {

    public static void main(String[] args) {
        GitConsole console = new GitConsole(new FullLogFormatter());

        List<Log> logs = List.of(
                new Log("397c670", "cleancoder", "Clean up", LocalDateTime.now()),
                new Log("07aa4d5", "cleancoder", "Fix NPE in Log", LocalDateTime.now()));

        console.printLog(logs);
    }
}
```

Advantages:

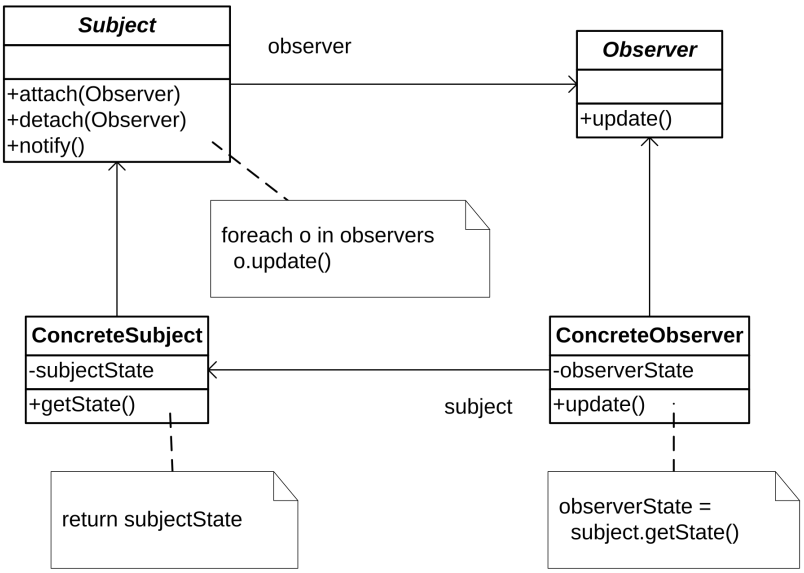
- **Reusability** and **decoupling of context and behaviour**. A family of algorithms is created, which can be used context-independently. On the one hand, new context objects can use the existing strategies.
- **Dynamic behaviour**. The behaviour of the context can be changed at runtime using appropriate setters.
- It is possible to choose from different implementations, thus increasing flexibility and reusability.
- **Multiple branches** can be avoided and this increases the overview of the code.
- **Alternative implementation**. Also, the same function (e.g., sorting) can be offered by different implementations, which differ in non-functional aspects (performance, memory requirements).

Disadvantages:

- Clients need to know the different strategies to choose between them and initialize the context.
  - Problem can be solved with a Factory. This allows the creation logic to be outsourced from the client to a factory.

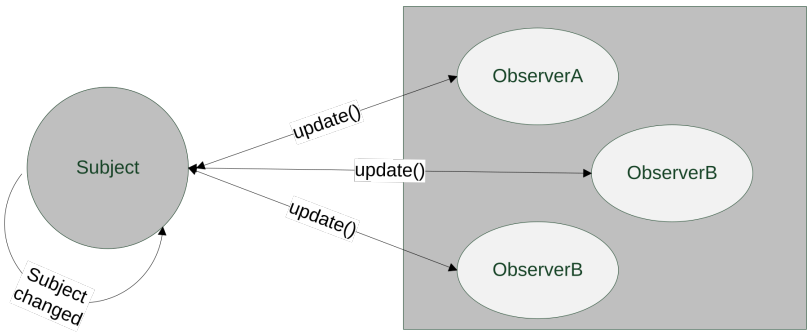
6.3.4 Observer

Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

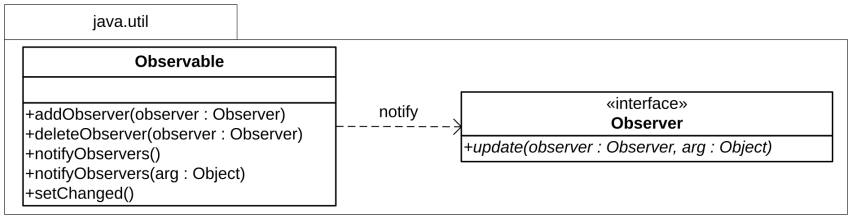


Use cases:

- When changing an object makes it is necessary to modify one or more objects.
  - GUIs (user changes data, new data must be updated in all GUI components).
  - MVC patterns (Model-View-Controller) in view-model communication.
  - Every second of a timer pulse, both the digital clock and the analog clock must be updated.
- Objects should notify other objects without knowing more about the object to be notified.
- Enables loose coupling of objects.



6.3.4.1 Example: `DataStore`



Screen

```
public class Screen implements Observer {
    public void update(Observable o, Object arg) {
        // act on the update
    }
}
```

DataStore

```
public class DataStore extends Observable {
    private String data;

    public String data() {
        return data;
    }

    public void data(String data) {
        this.data = data;
        // mark the observable as changed
        setChanged();
    }
}
```

Client

```
class Client {
    public static void main(String[] args) {
        Screen screen = new Screen();
        DataStore dataStore = new DataStore();

        // register observer
        dataStore.addObserver(screen);

        // do something with dataStore

        // send a notification
        dataStore.notifyObservers();
    }
}
```

The `Observable` class and the `Observer` interface have been deprecated in Java 9. The event model supported by `Observer` and `Observable` is quite limited, the order of notifications delivered by `Observable` is unspecified, and state changes are not in one-for-one correspondence with notifications.

6.3.4.2 Example: `Influencer`

Subject

```
public interface Subject {

    void register(Observer observer);

    void unregister(Observer observer);

    void notifyAllObservers(Object message);

}
```

Observer

```
interface Observer {
    void update(Object message);
}
```

Influencer

```
public class Influencer implements Subject {

    private final Queue<Observer> followers = new ConcurrentLinkedQueue<>();

    private final String name;

    public Influencer(String name) {
        this.name = name;
    }

    @Override
    public void register(Observer follower) {
        this.followers.add(follower);
    }

    @Override
    public void unregister(Observer follower) {
        this.followers.remove(follower);
    }

    @Override
    public void notifyAllObservers(Object message) {
        this.followers.forEach(follower -> follower.update(message));
    }
}
```

Follower

```
public class Follower implements Observer {

    private final String follower;

    public Follower(String follower) {
        this.follower = follower;
    }

    @Override
    public void update(Object message) {
        System.out.printf("%s received message: %s%n", this.follower, message);
    }
}
```

Client

```
class Client {

    public static void main(String[] args) {
        Influencer influencer = new Influencer("Java");

        influencer.register(new Follower("PHP"));
        influencer.register(new Follower("QBasic"));
        influencer.register(new Follower("TypeScript"));

        influencer.notifyAllObservers("Java rocks!");
    }
}
```

Advantages:

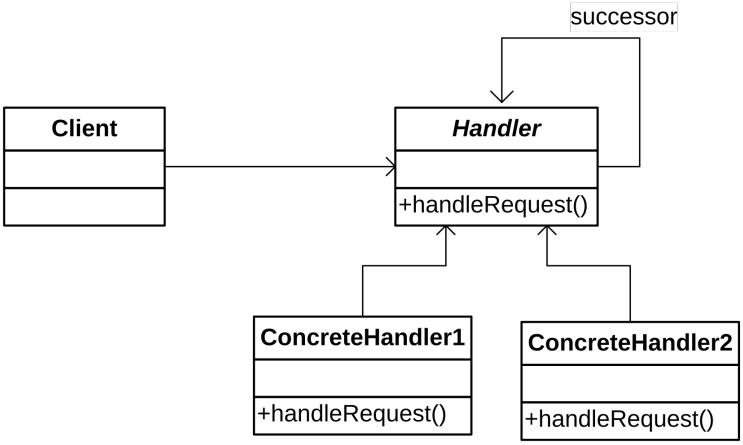
- **Consistency** of condition. Automatic adjustment of the observer state when the subject changes.
- **Reusability**. The Observer Pattern allows subject and observer to be varied independently.
- **Easy extension** of different observers that observe a single subject.

Disadvantages:

- **Update cascades and cycles**. In complex systems with many subjects and observers, update cascades can easily occur because the observers do not know about each other and cannot estimate the consequences of a single modification to a subject.

6.3.5 Chain of Responsibility

Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.



Use cases:

- Providing a low coupling between an object that should be handled by potential handler objects.
- Where a request can be handled by more than one object.
- Processing a request at runtime in dynamically sequential order.

6.3.5.1 Example: `Purchase`

Purchase

```
public record Purchase(double amount) {

}
```

PurchaseAuthorizeFlow

```
public interface PurchaseAuthorizeFlow {
    void next(Employee nextEmployee);
}
```

Employee

```
public interface Employee extends PurchaseAuthorizeFlow {
    void authorize(Purchase purchase);
}
```

AbstractEmployee

```
public abstract class AbstractEmployee implements Employee {

    private Employee nextEmployee;

    @Override
    public void next(Employee nextEmployee) {
        this.nextEmployee = nextEmployee;
    }

    public Employee next() {
        return this.nextEmployee;
    }
}
```

CTO

```
public class CTO extends AbstractEmployee {

    @Override
    public void authorize(Purchase purchase) {
        if (purchase.amount() > 100_000d) {
            System.out.println("CTO");
        } else {
            next().authorize(purchase);
        }
    }
}
```

VP

```
public class VP extends AbstractEmployee {

    @Override
    public void authorize(Purchase purchase) {
        if (purchase.amount() > 10_000d && purchase.amount() < 100_000d) {
            System.out.println("VP");
        } else {
            next().authorize(purchase);
        }
    }
}
```

TeamLead

```
public class TeamLead extends AbstractEmployee {

    @Override
    public void authorize(Purchase purchase) {
        if (purchase.amount() <= 10_000d) {
            System.out.println("TeamLead");
        } else {
            next().authorize(purchase);
        }
    }
}
```

Client

```
class Client {

    public static void main(String[] args) {
        Employee cto = new CTO();
        Employee vp = new VP();
        Employee teamLead = new TeamLead();

        teamLead.next(vp);
        vp.next(cto);

        teamLead.authorize(new Purchase(100));     // TeamLead
        teamLead.authorize(new Purchase(99_999));  // VP
        teamLead.authorize(new Purchase(500_000)); // CTO
    }
}
```

6.3.5.2 Example: `Authentication`

Authentication

```
public interface Authentication {
}
```

AuthenticationFilter

```
public interface AuthenticationFilter {
    boolean isAuthenticated(Authentication authentication);
    void setNextFilter(AuthenticationFilter nextFilter);
}
```

AbstractAuthenticationFilter

```
public abstract class AbstractAuthenticationFilter implements AuthenticationFilter {

    private AuthenticationFilter nextFilter;

    @Override
    public void setNextFilter(AuthenticationFilter nextFilter) {
        this.nextFilter = nextFilter;
    }

    public boolean nextFilter(Authentication authentication) {
        if (this.nextFilter != null) {
            return this.nextFilter.isAuthenticated(authentication);
        }

        return false;
    }
}
```

BasicAuthenticationFilter

```
public class BasicAuthenticationFilter extends AbstractAuthenticationFilter {

    @Override
    public boolean isAuthenticated(Authentication authentication) {
        if (authentication instanceof BasicAuthentication) {
            // logic
            return true;
        }

        return nextFilter(authentication);
    }
}
```

BasicAuthentication

```
public class BasicAuthentication implements Authentication {
}
```

BearerAuthenticationFilter

```
public class BearerAuthenticationFilter extends AbstractAuthenticationFilter {

    @Override
    public boolean isAuthenticated(Authentication authentication) {
        if (authentication instanceof BearerAuthentication) {
            // logic
            return true;
        }

        return nextFilter(authentication);
    }
}
```

BearerAuthentication

```
public class BearerAuthentication implements Authentication {
}
```

DigestAuthenticationFilter

```
public class DigestAuthenticationFilter extends AbstractAuthenticationFilter {

    @Override
    public boolean isAuthenticated(Authentication authentication) {
        if (authentication instanceof DigestAuthentication) {
            // logic
            return true;
        }

        return nextFilter(authentication);
    }
}
```

DigestAuthentication

```
public class DigestAuthentication implements Authentication {
}
```

Client

```
class Client {

    public static void main(String[] args) {
        AuthenticationFilter digestFilter = new DigestAuthenticationFilter();
        AuthenticationFilter bearerFilter = new BearerAuthenticationFilter();
        AuthenticationFilter basicFilter = new BasicAuthenticationFilter();

        digestFilter.setNextFilter(bearerFilter);
        bearerFilter.setNextFilter(basicFilter);

        System.out.println(digestFilter.isAuthenticated(new BearerAuthentication()));  // true
    }
}
```

Advantages:

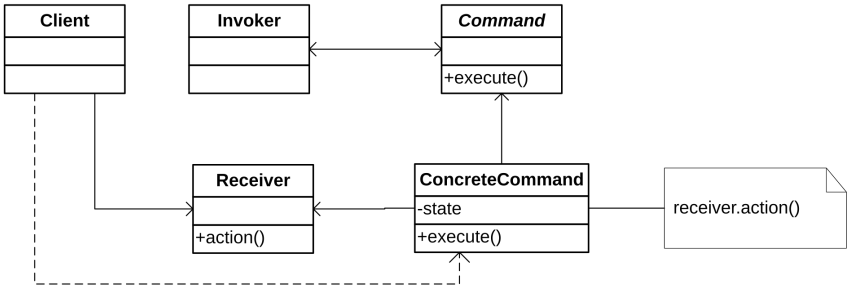
- The object does not need to know the structure of the chain, which decouples the sender of the request and its recipients.
- Simplifies your object as it does not need to know the structure of the chain and does not need to hold a direct reference to its members.
- Allows responsibilities to be added and removed dynamically by changing the members of the order of the chain.

Disadvantages:

- It is not guaranteed that objects are handled in the chain's flow, but this can also be seen as an advantage.
- It can be difficult to observe and debug the runtime properties.

6.3.6 Command

Encapsulates a request as an object, thereby letting you parameterize clients with different requests, queue, or log requests, and support undoable operations.



Use cases:

- When you want to **queue** operations or **schedule** their execution.
- Implementing **undo** or **redo** operations.

6.3.6.1 Example: `FileSystem`

Command

```
public interface Command {
    void execute(FileSystem fs);
}
```

CreateCommand

```
public class CreateCommand implements Command {

    private final File file;

    public CreateCommand(File file) {
        this.file = file;
    }

    @Override
    public void execute(FileSystem fs) {
        fs.create(this.file);
    }
}
```

DeleteCommand

```
public class DeleteCommand implements Command {

    private final File file;

    public DeleteCommand(File file) {
        this.file = file;
    }

    @Override
    public void execute(FileSystem fs) {
        fs.delete(this.file);
    }
}
```

MoveCommand

```
public class MoveCommand implements Command {

    private final File source;
    private final File target;

    public MoveCommand(File source, File target) {
        this.source = source;
        this.target = target;
    }

    @Override
    public void execute(FileSystem fs) {
        fs.move(this.source, this.target);
    }
}
```

FileSystem

```
public class FileSystem {

    public void create(File file) {
        System.out.printf("Create %s%n",
                file.getName());
    }

    public void delete(File file) {
        System.out.printf("Delete %s%n",
                file.getName());
    }

    public void move(File source, File target) {
        System.out.printf("Move %s to %s%n",
                source.getName(),
                target.getName());
    }
}
```

BatchFileSystemExecutor

```
public class BatchFileSystemExecutor {

    private final FileSystem fs;

    public BatchFileSystemExecutor(FileSystem fs) {
        this.fs = fs;
    }

    public void execute(List<Command> commands) {
        commands.forEach(c -> c.execute(this.fs));
    }
}
```

Client

```
class Client {

    public static void main(String[] args) {
        FileSystem fs = new FileSystem();
        BatchFileSystemExecutor executor = new BatchFileSystemExecutor(fs);

        List<Command> commands = List.of(
                new CreateCommand(new File("file.tmp")),
                new DeleteCommand(new File("file.tmp")),
                new CreateCommand(new File("secret.txt")),
                new MoveCommand(new File("secret.txt"), new File("topsecret.txt")));

        executor.execute(commands);
    }
}
```

Output

```
Create file.tmp
Delete file.tmp
Create secret.txt
Move secret.txt to topsecret.txt
```

6.3.6.2 Example: `Television`

Command

```
public interface Command {
    void execute();
}
```

TurnOn

```
public class TurnOn implements Command {

    private final Television device;

    public TurnOn(Television device) {
        this.device = device;
    }

    @Override
    public void execute() {
        this.device.on();
    }
}
```

TurnOff

```
public class TurnOff implements Command {

    private final Television device;

    public TurnOff(Television device) {
        this.device = device;
    }

    @Override
    public void execute() {
        this.device.off();
    }
}
```

VolumeUp

```
public class VolumeUp implements Command {

    private final Television device;

    public VolumeUp(Television device) {
        this.device = device;
    }

    @Override
    public void execute() {
        this.device.volumeUp();
    }
}
```

VolumeDown

```
public class VolumeDown implements Command {

    private final Television device;

    public VolumeDown(Television device) {
        this.device = device;
    }

    @Override
    public void execute() {
        this.device.volumeDown();
    }
}
```

Television

```
public class Television {

    private int volume = 0;

    public void on() {
        System.out.println("TV is on");
    }

    public void off() {
        System.out.println("TV is off");
    }

    public void volumeUp() {
        this.volume++;

        System.out.println("Volume: " + this.volume);
    }

    public void volumeDown() {
        this.volume--;

        System.out.println("Volume: " + this.volume);
    }
}
```

DeviceButton

```
public class DeviceButton {

    private Command command;

    public void command(Command command) {
        this.command = command;
    }

    public void press() {
        this.command.execute();
    }
}
```

Client

```
class Client {

    public static void main(String[] args) {
        Television television = new Television();

        DeviceButton power = new DeviceButton();
        DeviceButton volume = new DeviceButton();

        power.command(new TurnOn(television));
        power.press();

        volume.command(new VolumeUp(television));
        volume.press();
        volume.press();
        volume.press();

        volume.command(new VolumeDown(television));
        volume.press();

        power.command(new TurnOff(television));
        power.press();
    }
}
```

Output

```
TV is on
Volume: 1
Volume: 2
Volume: 3
Volume: 2
TV is off
```

Advantages:

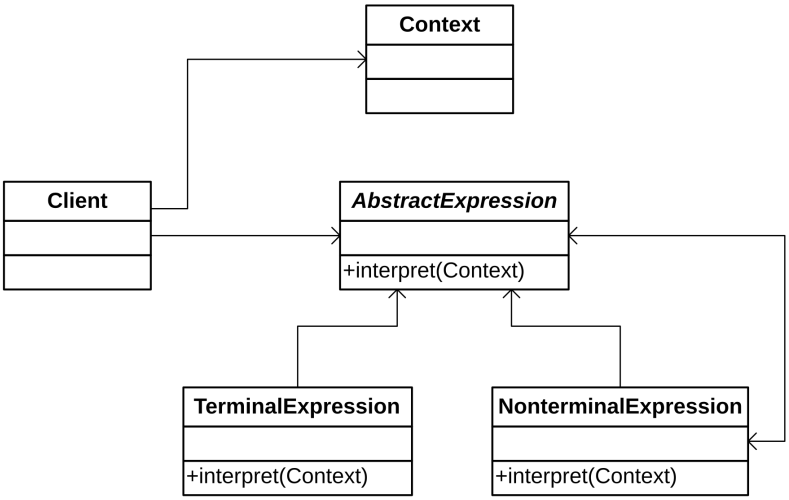
- New commands can be **added easily** without changing existing code.
- Possibility of implementing **undo** and **redo** operation.
- You can implement a **sequence** of operations.

Disadvantages:

- **Large number** of classes and objects working together to achieve the desired goal.

6.3.7 Interpreter

Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.



Use cases:

- When the grammar is **simple**. A class hierarchy can be designed to represent the set of grammar rules with every class in the hierarchy representing a separate grammar rule.
- Representing a simple grammar as an **abstract syntax tree** like structure.

6.3.7.1 Example: `HexBinary`

Expression

```
public abstract class Expression {

    private final int value;

    protected Expression(int value) {
        this.value = value;
    }

    public int value() {
        return this.value;
    }

    abstract String interpret(Context context);
}
```

IntToBinaryExpression

```java
public final class IntToBinaryExpression extends Expression {

    public IntToBinaryExpression(int value) {
        super(value);
    }

    @Override
    public String interpret(Context context) {
        return Integer.toBinaryString(value());
    }
}
```

IntToHexExpression

```java
public final class IntToHexExpression extends Expression {

    public IntToHexExpression(int value) {
        super(value);
    }

    @Override
    public String interpret(Context context) {
        return Integer.toHexString(value());
    }
}
```

Context

```java
public final class Context {

    private final String expression;

    public Context(String expression) {
        this.expression = Objects.requireNonNull(expression);
    }

    public String expression() {
        return this.expression;
    }

    public boolean isToHex() {
        return this.expression.contains("Hexadecimal");
    }

    public boolean isToBinary() {
        return this.expression.contains("Binary");
    }

    public int number() {
        return Integer.parseInt(this.expression.substring(0, this.expression.indexOf(' ')));
    }
}
```

Client

```
class Client {

    public static void main(String[] args) {
        print("42 in Binary");
        print("42 in Hexadecimal");
    }

    private static void print(String expression) {
        System.out.printf("%s = %s%n", expression, interpret(expression));
    }

    private static String interpret(String expression) {
        Context context = new Context(expression);
        Expression exp;

        if (context.isToHex()) {
            exp = new IntToHexExpression(context.number());
        } else if (context.isToBinary()) {
            exp = new IntToBinaryExpression(context.number());
        } else {
            return context.expression();
        }

        return exp.interpret(context);
    }
}
```

Output

```
42 in Binary = 101010
42 in Hexadecimal = 2a
```

6.3.7.2 Example: `Calculator`

Expression

```
interface Expression {
    int interpret();
}
```

Number

```
public final class Number implements Expression {

    private final int number;

    public Number(String number) {
        this.number = Integer.parseInt(number);
    }

    @Override
    public int interpret() {
        return this.number;
    }
}
```

Minus

```
public final class Minus implements Expression {

    private final Expression left;
    private final Expression right;

    public Minus(Expression left, Expression right) {
        this.left = left;
        this.right = right;
    }

    @Override
    public int interpret() {
        return this.left.interpret() - this.right.interpret();
    }
}
```

Plus

```
public final class Plus implements Expression {

    private final Expression left;
    private final Expression right;

    public Plus(Expression left, Expression right) {
        this.left = left;
        this.right = right;
    }

    @Override
```

```
    public int interpret() {
        return this.left.interpret() + this.right.interpret();
    }
}
```

Evaluator

```
public final class Evaluator {

    private final String expression;

    public Evaluator(String expression) {
        this.expression = Objects.requireNonNull(expression);
    }

    public String expression() {
        return this.expression;
    }

    public int evaluate() {
        Expression last = null;
        String splitted[] = this.expression.split(" ");

        for (int i = 0, splitted.length; i < length; i++) {
            if ("+".equals(splitted[i])) {
                last = new Plus(last, new Number(splitted[i + 1]));
                i++;
            } else if ("-".equals(splitted[i])) {
                last = new Minus(last, new Number(splitted[i + 1]));
                i++;
            } else {
                last = new Number(splitted[i]);
            }
        }

        return last.interpret();
    }
}
```

Client

```
class Client {

    public static void main(String[] args) {
        Evaluator evaluator = new Evaluator("5 - 3 + 29 - 1");

        System.out.printf("%s = %s%n",
                evaluator.expression(),
                evaluator.evaluate());
    }
}
```

Output

```
5 - 3 + 29 - 1 = 30
```

Advantages:

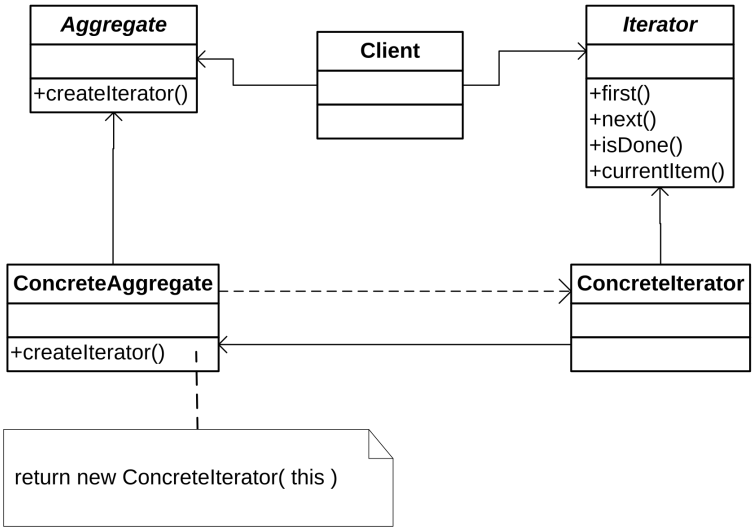
- Easy to change and extend the grammar. Existing expressions can be inherited and modified as required.

Disadvantages:

- Complex grammars are **difficult to maintain** and could become **unmanageable**.

6.3.8 Iterator

Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.



Use cases:

- Access aggregate objects without exposing their internal representation.
- Support multiple traversals of aggregate objects.
- Provide a uniform interface for traversing different aggregate structures.

An iterator can be designed as either an **internal** iterator or as **external** iterator.

Internal Iterators:

- The collection itself provides methods that allow a client to visit various objects within the collection. For example, the `java.util.ResultSet` class contains the data and also provides methods like `next()` to navigate through the item list.
- There can only be one iterator in a collection at a time.
- The collection must maintain or store the state of the iteration.

External Iterators:

- The iteration functionality is separate from the collection and resides in another object called an iterator. Normally, the collection itself returns a corresponding iterator object to the client. For example, the `java.util.Vector`, has its iterator defined in the form of a separate object of the type `Enumeration`.
- There can be multiple iterators for a given collection at any point in time.
- The overhead associated with storing the iteration state is not associated with the collection. It lies within the exclusive iterator object.

6.3.8.1 Example: `Cars - intern`

Car

```
public record Car(String name) {

}
```

Client

```
class Client {

    public static void main(String[] args) {
        List<Car> cars = List.of(new Car("Car 1"), new Car("Car 2"), new Car("Car 3"), new Car("Car 4"\
));

        cars.forEach(c -> System.out.println(c.name()));
    }
}
```

Output

```
Car 1
Car 2
Car 3
Car 4
```

6.3.8.2 Example: `Cars - extern`

Car

```
public record Car(String name) {

}
```

CarIterator

```
public class CarIterator implements java.util.Iterator<Car> {

    private final Car[] cars;
    private int index = 0;

    public CarIterator(Car[] cars) {
        this.cars = cars;
    }

    @Override
    public Car next() {
        return this.cars[this.index++];
    }

    @Override
    public boolean hasNext() {
        if (this.index >= this.cars.length || this.cars[this.index] == null) {
            return false;
        }

        return true;
    }
}
```

Client

```
class Client {

    public static void main(String[] args) {
        Car[] cars = new Car[4];

        cars[0] = new Car("Car 1");
        cars[1] = new Car("Car 2");
        cars[2] = new Car("Car 3");
        cars[3] = new Car("Car 4");

        Iterator<Car> iterator = new CarIterator(cars);

        while (iterator.hasNext()) {
            Car car = iterator.next();
            System.out.println(car.name());
        }
    }
}
```

Output

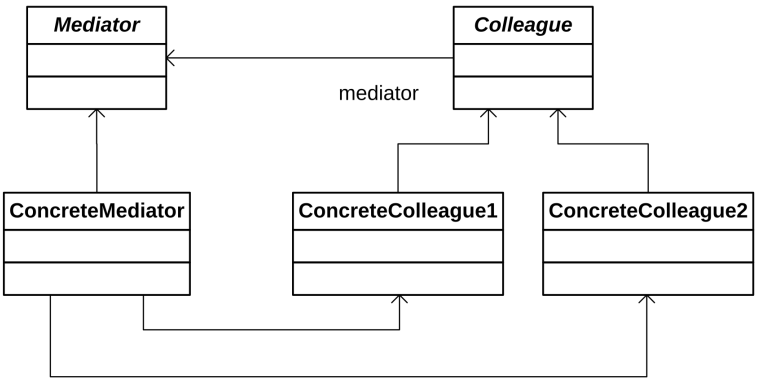
```
Car 1
Car 2
Car 3
Car 4
```

Advantages:

- **Parallel iteration** over the same collection because each iterator object contains its own iteration state (external iterator).
- Allow iterating through an aggregate object **without exposing** its internal representation.
- The traversal task is done by the iterator, which **simplifies** the aggregation class.

6.3.9 Mediator

Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.



Use cases:

- Many objects communicate in well-defined but complex ways. The resulting dependencies are unstructured and difficult to understand.
- Reusing an object is difficult because it interacts and communicates with many other objects.
- A behavior that is distributed across several classes should be changeable without much subclassing.

6.3.9.1 Example: `Chat`

ChatMediator

```
interface ChatMediator {

    void sendMessage(User user, String message);

    void addUser(User user);
}
```

Chat

```java
public final class Chat implements ChatMediator {

    private final List<User> users;

    public Chat() {
        this.users = new ArrayList<>();
    }

    @Override
    public void addUser(User user) {
        this.users.add(user);
    }

    @Override
    public void sendMessage(User fromUser, String message) {
        for (User user : this.users) {
            if (user != fromUser) {
                user.receive(message);
            }
        }
    }
}
```

User

```java
public class User {

    private final ChatMediator mediator;
    private final String name;

    public User(ChatMediator mediator, String name) {
        this.mediator = mediator;
        this.name = name;
    }

    public ChatMediator mediator() {
        return this.mediator;
    }

    public String name() {
        return this.name;
    }

    public void send(String message) {
        System.out.println(name() + ": Sending Message=" + message);
        mediator().sendMessage(this, message);
    }

    public void receive(String message) {
        System.out.println(name() + ": Received Message:" + message);
    }
}
```

LoggedInUser

```
public class LoggedInUser extends User {

    public LoggedInUser(ChatMediator mediator, String name) {
        super(mediator, name);
    }

}
```

AnonymousUser

```
public class AnonymousUser extends User {

    public AnonymousUser(ChatMediator mediator) {
        super(mediator, "Anonymous");
    }

}
```

Client

```
class Client {

    public static void main(String[] args) {
        ChatMediator mediator = new Chat();
        User user1 = new LoggedInUser(mediator, "Lia");
        User user2 = new LoggedInUser(mediator, "Timo");
        User user3 = new LoggedInUser(mediator, "Matheo");
        User user4 = new LoggedInUser(mediator, "Luisa");
        User user5 = new AnonymousUser(mediator);

        mediator.addUser(user1);
        mediator.addUser(user2);
        mediator.addUser(user3);
        mediator.addUser(user4);
        mediator.addUser(user5);

        user1.send("Hi to everybody!");
    }
}
```

Output

```
Lia: Sending Message=Hi to everybody!
Timo: Received Message:Hi to everybody!
Matheo: Received Message:Hi to everybody!
Luisa: Received Message:Hi to everybody!
Anonymous: Received Message:Hi to everybody!
```

6.3.9.2 Example: Aircraft

AircraftMediator

```java
public interface AircraftMediator {

    void registerRunway(Runway runway);

    String allotRunway(Aircraft aircraft);

    void releaseRunway(Aircraft aircraft);
}
```

AircraftTrafficControlRoom

```java
public final class AircraftTrafficControlRoom implements AircraftMediator {

    private final LinkedList<Runway> availableRunways;
    private final Map<Aircraft, Runway> aircraftRunways;

    public AircraftTrafficControlRoom() {
        this.availableRunways = new LinkedList<>();
        this.aircraftRunways = new HashMap<>();
    }

    @Override
    public void registerRunway(Runway runway) {
        this.availableRunways.add(runway);
    }

    @Override
    public String allotRunway(Aircraft aircraft) {
        Runway nextAvailableRunway = null;

        if (!this.availableRunways.isEmpty()) {
            nextAvailableRunway = this.availableRunways.removeFirst();
            this.aircraftRunways.put(aircraft, nextAvailableRunway);
        }

        return nextAvailableRunway == null ? null : nextAvailableRunway.name();
    }

    @Override
    public void releaseRunway(Aircraft aircraft) {
        if (this.aircraftRunways.containsKey(aircraft)) {
            Runway runway = this.aircraftRunways.remove(aircraft);
            this.availableRunways.add(runway);
        }
    }
}
```

AircraftColleague

```java
public interface AircraftColleague {

    void startLanding();

    void finishLanding();
}
```

Aircraft

```java
public final class Aircraft implements AircraftColleague {

    private final AircraftMediator mediator;
    private final String flightName;

    public Aircraft(AircraftMediator mediator, String flightName) {
        this.mediator = mediator;
        this.flightName = flightName;
    }

    @Override
    public void startLanding() {
        String runway = this.mediator.allotRunway(this);

        if (runway == null) {
            System.out.println("Due to traffic, there's a delay in landing of " + this.flightName);
        } else {
            System.out.println("Currently landing " + this.flightName + " on " + runway);
        }
    }

    @Override
    public void finishLanding() {
        System.out.println(this.flightName + " landed successfully");
        this.mediator.releaseRunway(this);
    }
}
```

Runway

```java
public record Runway(String name) {

}
```

Client

```
class Client {

    public static void main(String[] args) {
        AircraftMediator mediator = new AircraftTrafficControlRoom();
        mediator.registerRunway(new Runway("Runway A"));

        AircraftColleague wrightFlight = new Aircraft(mediator, "Wright Flight");
        AircraftColleague airbusA320 = new Aircraft(mediator, "Airbus A320");

        wrightFlight.startLanding();
        airbusA320.startLanding();
        wrightFlight.finishLanding();
    }
}
```

Output

```
Currently landing Wright Flight on Runway A
Due to traffic, there's a delay in landing of Airbus A320
Wright Flight landed successfully
```

Advantages:

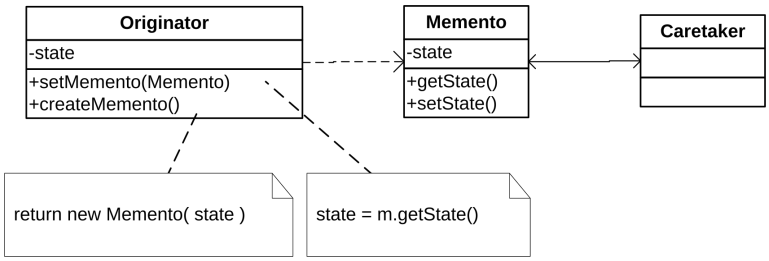
- The mediator is the **only object** that knows about all other objects.
- It becomes easier to **change the behavior** of object relationships by replacing the mediator with one of its subclasses with extended or changed functionality.
- Moving the dependencies between objects out of the individual objects leads to better **reusability** of the objects.
- Since the objects do not have to refer to each other directly, it is **easier to test**.
- The resulting low coupling makes it possible to **change individual classes** without affecting other classes.

Disadvantages:

- You have to be careful not to create a so-called **god object**, which happens when the mediator class becomes too big.

6.3.10 Memento

Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.



Use cases:

- For designing a mechanism to capture and store the state of an object so, when needed, the object can be put back to this previous state.

6.3.10.1 Example: `Editor`

Editor

```
1  public interface Editor {
2
3      void setContent(String content);
4
5      void setFontSize(int fontSize);
6
7      void setFontName(String fontName);
8
9      void save();
10
11     void restore();
12 }
```

EditorState

```
1  public interface EditorState {
2
3      String getContent();
4
5      int getFontSize();
6
7      String getFontName();
8
9  }
```

CleanCodeEditor

```java
public final class CleanCodeEditor implements Editor {

    private final History history = new History();

    private String content;
    private int fontSize;
    private String fontName;

    public CleanCodeEditor(String content, int fontSize, String fontName) {
        this.content = content;
        this.fontName = fontName;
        this.fontSize = fontSize;
    }

    public String getContent() {
        return this.content;
    }

    public int getFontSize() {
        return this.fontSize;
    }

    public String getFontName() {
        return this.fontName;
    }

    @Override
    public void setContent(String content) {
        this.content = content;
    }

    @Override
    public void setFontSize(int fontSize) {
        this.fontSize = fontSize;
    }

    @Override
    public void setFontName(String fontName) {
        this.fontName = fontName;
    }

    @Override
    public void save() {
        EditorState state = new CleanCodeEditorState(this.content, this.fontSize, this.fontName);
        this.history.push(state);
    }

    @Override
    public void restore() {
        EditorState state = this.history.pop();
        this.content = state.getContent();
        this.fontName = state.getFontName();
        this.fontSize = state.getFontSize();
    }
}
```

CleanCodeEditorState

```java
public final class CleanCodeEditorState implements EditorState {

    private final String content;
    private final int fontSize;
    private final String fontName;

    public CleanCodeEditorState(String content, int fontSize, String fontName) {
        this.content = content;
        this.fontName = fontName;
        this.fontSize = fontSize;
    }

    @Override
    public String getContent() {
        return this.content;
    }

    @Override
    public int getFontSize() {
        return this.fontSize;
    }

    @Override
    public String getFontName() {
        return this.fontName;
    }
}
```

History

```java
public final class History {

    private final Stack<EditorState> history;

    public History() {
        this.history = new Stack<>();
    }

    public void push(EditorState state) {
        this.history.push(state);
    }

    public EditorState pop() {
        return this.history.pop();
    }
}
```

Client

```
class Client {

    public static void main(String[] args) {
        Editor editor = new CleanCodeEditor("Java Clean Code", 12, "Courier New");
        editor.save();

        editor.setContent("Java");
        editor.save();

        editor.restore();
        System.out.println(((CleanCodeEditor) editor).getContent());

        editor.restore();
        System.out.println(((CleanCodeEditor) editor).getContent());
    }
}
```

Output

```
Java
Java Clean Code
```

6.3.10.2 Example: `Balance`

Originator

```java
public class Originator {

    private Double balance;

    public void setBalance(Double balance) {
        this.balance = balance;
    }

    public Double getBalance() {
        return this.balance;
    }

    public Memento save() {
        return new Memento(this.balance);
    }

    public void restore(Memento memento) {
        this.balance = memento.balance();
    }
}
```

Memento

```java
public record Memento(Double balance) {

}
```

CareTaker

```java
public final class CareTaker {

    private final Stack<Memento> history;

    public CareTaker() {
        this.history = new Stack<>();
    }

    public void push(Memento state) {
        this.history.push(state);
    }

    public Memento pop() {
        return this.history.pop();
    }
}
```

Client

```
class Client {

    public static void main(String[] args) {
        CareTaker careTaker = new CareTaker();
        Originator originator = new Originator();

        originator.setBalance(40.11d);
        careTaker.push(originator.save());
        System.out.println(originator.getBalance());

        originator.setBalance(100.11d);
        careTaker.push(originator.save());
        System.out.println(originator.getBalance());

        originator.restore(careTaker.pop());
        System.out.println(originator.getBalance());

        originator.restore(careTaker.pop());
        System.out.println(originator.getBalance());
    }
}
```

Output

```
40.11
100.11
100.11
40.11
```

Advantages:

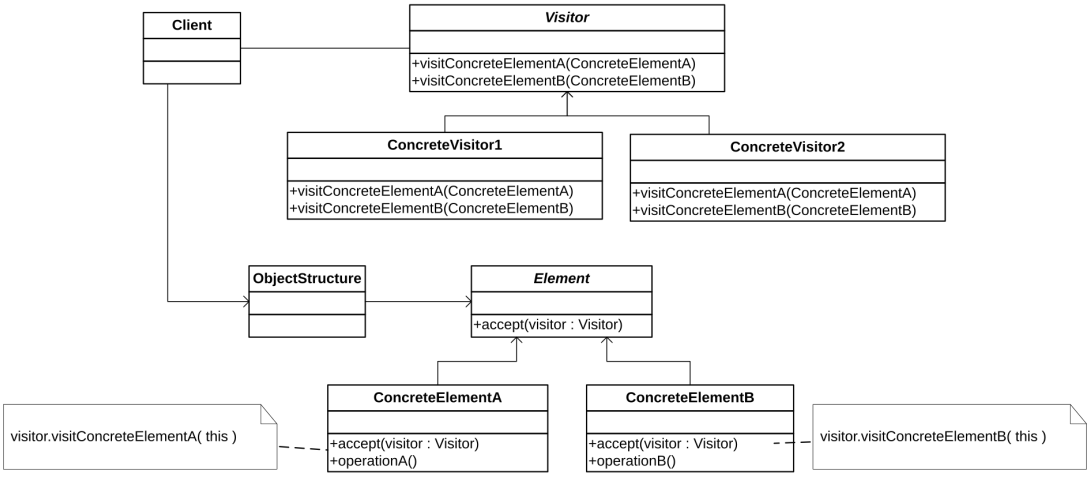
- You always can **discard changes** and **restore** them to a stable state.
- **Easy restore techniques** are provided by this pattern.

Disadvantages:

- If you allow a huge amount of history entries it can get a **memory problem**.
- Saving the states will **reduce** the overall **performance** of the application.

6.3.11 Visitor

Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.



Use cases:

- Lets you define a new operation without changing the classes of the elements on which it operates.
- Similar operations must be performed on objects of different types across a heterogeneous class hierarchy.

6.3.11.1 Example: `Fridge`

FridgeElement

```
public interface FridgeElement {
    void accept(Visitor visitor);
}
```

Visitor

```
public interface Visitor {

    void visit(Beer beer);

    void visit(Milk milk);

    void visit(Butter butter);

}
```

Beer

```
public class Beer implements FridgeElement {

    @Override
    public void accept(Visitor visitor) {
        visitor.visit(this);
    }

    LocalDate bestBeforeDate() {
        return LocalDate.of(2022, 1, 28);
    }
}
```

Butter

```
public class Butter implements FridgeElement {

    @Override
    public void accept(Visitor visitor) {
        visitor.visit(this);
    }

    LocalDate expiryDate() {
        return LocalDate.of(2021, 10, 2);
    }
}
```

Milk

```
public class Milk implements FridgeElement {

    @Override
    public void accept(Visitor visitor) {
        visitor.visit(this);
    }

    Date bestBeforeDate() {
        return new Date(2021 - 1900, Calendar.JANUARY, 1);
    }
}
```

BestBeforeDateVisitor

```
public class BestBeforeDateVisitor implements Visitor {

    private final LocalDate compareDate;

    public BestBeforeDateVisitor(LocalDate compareDate) {
        this.compareDate = compareDate;
    }

    @Override
    public void visit(Beer beer) {
        checkBestBeforeDate(beer, beer.bestBeforeDate()); // method differ
```

```
    }

    @Override
    public void visit(Milk milk) {
        checkBestBeforeDate(milk, milk.bestBeforeDate() // method differ
                .toInstant()
                .atZone(ZoneId.systemDefault())
                .toLocalDate());
    }

    @Override
    public void visit(Butter butter) {
        checkBestBeforeDate(butter, butter.expiryDate()); // method differ
    }

    private void checkBestBeforeDate(FridgeElement element, LocalDate bestBeforeDate) {
        if (bestBeforeDate.isAfter(this.compareDate)) {
            System.out.println(element.getClass().getSimpleName());
        }
    }
}
```

Client

```
class Client {

    public static void main(String[] args) {
        List<FridgeElement> fridge = List.of(
                new Beer(),
                new Beer(),
                new Milk(),
                new Butter());

        Visitor visitor = new BestBeforeDateVisitor(LocalDate.of(2021, 1, 1));

        System.out.println("Over the best before date:");

        for (FridgeElement element : fridge) {
            element.accept(visitor);
        }
    }
}
```

Output

```
Over the best before date:
Beer
Beer
Butter
```

6.3.11.2 Example: Figures

Figure

```
interface Figure {
    <T> T accept(Visitor<T> visitor);
}
```

Visitor

```
interface Visitor<T> {

    T visit(Square square);

    T visit(Circle circle);

    T visit(Rectangle rectangle);
}
```

Circle

```
public class Circle implements Figure {

    private final double radius;

    public Circle(double radius) {
        this.radius = radius;
    }

    public double radius() {
        return this.radius;
    }

    @Override
    public <T> T accept(Visitor<T> visitor) {
        return visitor.visit(this);
    }
}
```

Rectangle

```
public class Rectangle implements Figure {

    private final double weight;
    private final double height;

    public Rectangle(double weight, double height) {
        this.weight = weight;
        this.height = height;
    }

    public double weight() {
        return this.weight;
    }

```

```
    public double height() {
        return this.height;
    }

    @Override
    public <T> T accept(Visitor<T> visitor) {
        return visitor.visit(this);
    }
}
```

Square

```
public class Square implements Figure {

    private final double side;

    public Square(double side) {
        this.side = side;
    }

    public double side() {
        return this.side;
    }

    @Override
    public <T> T accept(Visitor<T> visitor) {
        return visitor.visit(this);
    }
}
```

AreaVisitor

```
public class AreaVisitor implements Visitor<Double> {

    @Override
    public Double visit(Square square) {
        return square.side() * square.side();
    }

    @Override
    public Double visit(Circle circle) {
        return Math.PI * circle.radius() * circle.radius();
    }

    @Override
    public Double visit(Rectangle rectangle) {
        return rectangle.height() * rectangle.weight();
    }
}
```

PerimeterVisitor

```java
public class PerimeterVisitor implements Visitor<Double> {

    @Override
    public Double visit(Square element) {
        return 4 * element.side();
    }

    @Override
    public Double visit(Circle element) {
        return 2 * Math.PI * element.radius();
    }

    @Override
    public Double visit(Rectangle element) {
        return 2 * element.height() + 2 * element.height();
    }
}
```

Client

```java
class Client {

    public static void main(String[] args) {
        List<Figure> figures = List.of(
                new Circle(3),
                new Square(4),
                new Rectangle(2, 8));

        calculateArea(figures);
        calculatePerimeter(figures);
    }

    private static void calculateArea(List<Figure> figures) {
        Visitor<Double> visitor = new AreaVisitor();

        double totalArea = figures.stream()
                .mapToDouble(f -> f.accept(visitor))
                .sum();

        System.out.printf("Total area: %f %n", totalArea);
    }

    private static void calculatePerimeter(List<Figure> figures) {
        Visitor<Double> visitor = new PerimeterVisitor();

        double totalPerimeter = figures.stream()
                .mapToDouble(f -> f.accept(visitor))
                .sum();

        System.out.printf("Total perimeter: %f%n", totalPerimeter);
    }
}
```

Output

```
Total area: 60,274334
Total perimeter: 66,849556
```

Advantages:

- New operations can be **added easily**.
- The logic of the operations is **centralized** in the visitor and not dispersed.
- The visitor pattern allows the operation to be defined **without changing the class** of any of the objects in the collection.

Disadvantages:

- The implementation of the visitor concrete classes could **break the encapsulation** of the visited objects. Due to the fact, that a new visitor class needs access to the public members of these objects.